The New and Improved

Flask Mega-Tutorial



Miguel Grinberg

目錄

Flask 大型教程 2017

> 来源：luhuisicnu/The-Flask-Mega-Tutorial-zh

说明

本教程翻译自Miguel Grinberg的blog的2017年新版The Flask Mega-Tutorial教程，以供英语能力较弱的开发人员参考。感谢Miguel Grinberg！

全部二十三章都已完成翻译，如果有任何版权问题，请联系luhuisicnu@163.com。

如果有任何技术疑问，欢迎加入QQ群（484327418）讨论。

本文翻译自The Flask Mega-Tutorial Part I: Hello, World!

一趟愉快的学习之旅即将开始，跟随它你将学会用Python和Flask来创建Web应用。上面的视频包含了整个教程的内容预览（译者注：视频见原文）。通过学习本章内容，你将学会如何创建一个Flask项目，并在自己的电脑上运行一个简单的Flask Web应用。

教程中所有的代码示例都托管在GitHub上。虽然直接从GitHub下载代码可以节省写代码的步骤，但是我强烈建议你至少在前几章自己动手书写这些代码。一旦你熟悉了Flask和示例应用，一些繁琐重复的代码就可以直接从GitHub复制了。

在每章的开头，我都将提供三个GitHub的链接来帮助你顺畅地学习本章的内容。点击**Browse**链接会打开GitHub上Microblog项目在本章的对应代码库页面，不会包含之后章节的任何新增代码。而**Zip**链接则提供了这份代码库的zip打包文件的下载地址。如果点击**Diff**链接，打开的将会是本章节的代码变更信息。

本章的GitHub链接为: Browse, Zip, Diff.

安装Python

你说你还没有安装Python？那还等什么！立马安装吧。如果操作系统默认没有提供Python安装包，可以从Python官方网站下载。如果你使用Microsoft Windows操作系统并且打算使用WSL或者Cygwin，需要注意，不要在上面使用Windows版本的Python，而要使用类Unix版本，比如从Ubuntu获取（对应WSL）或从Cygwin上获取。

为了验证Python是否正确安装，你可以打开一个终端窗口并输入`python3`（如果不存在这个命令，那就输入`python`）。预期的输出如下：

```
$ python3
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```

Python解释器中，光标不断闪烁，等待着你输入Python语句。在未来的章节中，你可以充分体会到交互式解释器的魅力。至少现在它能够帮你确认Python已经成功安装的事实。可以输入`exit()`并回车来退出交互式解释器。在Linux和Mac OS X操作系统上，按下快捷键Ctrl-D也可以快速退出交互式解释器。在Windows操作系统上，则是通过按下Ctrl-Z后跟上Enter快捷键来快速退出。

安装Flask

下一步开始安装Flask，在这之前我要告诉你安装Python三方包的最佳实践。

Python将所有三方包托管到一个公共仓库，任何人都能从这个公共仓库下载并安装所有的三方包。Python将三方包公共仓库命名为PyPI以表示Python Package Index的缩写(被一些人戏称为"cheese shop")。从PyPI上安装三方包非常简单，Python专门提供了一个名为 `pip` 的工具来解决这个问题（Python2.7中不含 `pip` 工具，需要单独安装）。

安装三方包时，使用 `pip` 命令如下：

```
$ pip install <package-name>
```

有趣的是，这个方法在大多数情况下不适用。假如Python解释器是全局安装的，所有用户都能使用，那么普通用户则没有权限来修改它，因此只能用管理员账户来执行安装操作。即使忽略操作的复杂性，使用这种全局安装的方式会发生什么？ `pip` 工具从PyPI上下载三方包并安装到全局Python目录下，即刻起，所有Python脚本都可以访问到这个三方包。想象这样一个场景，你之前用当时的最新版本Flask——0.11版本的Flask开发了一个Web应用，现在Flask已经更新到了0.12版本，你想要使用0.12版本的Flask开发第二个Web应用。但是，如果将Flask从0.11版本升级到0.12版本可能会导致第一个Web应用出现故障。解决这个问题的方法最好不过为旧Web应用安装和使用Flask0.11版本，为新Web应用安装和使用Flask0.12版本。

为了解决维护不同应用程序对应不同版本的问题，Python使用了虚拟环境的概念。 虚拟环境是Python解释器的完整副本。在虚拟环境中安装三方包时只会作用到虚拟环境，全局Python解释器不受影响。 那么，就为每个应用程序安装各自的虚拟环境吧。 虚拟环境还有一个好处，即它们由创建它们的用户所拥有，所以不需要管理员帐户。

我们先创建项目目录，我将这个应用命名为*microblog*：

```
$ mkdir microblog
$ cd microblog
```

如果你正在使用Python3，虚拟环境已经成为内置模块，可以直接通过如下命令来创建它：

```
$ python3 -m venv venv
```

> 译者注：这个命令不一定能够执行成功，比如译者在Ubuntu16.04环境下执行，提示需要先安装对应的依赖。 `sudo apt-get install python3-venv`

使用这个命令来让Python运行 `venv` 包，它会创建一个名为 `venv` 的虚拟环境。 命令中的第一个“venv”是Python虚拟环境包的名称，第二个是要用于这个特定环境的虚拟环境名称。 如果你觉得这样很混乱，可以用你自定义的虚拟环境名字替换第二个 `venv` 。我习惯在项目目录中创建了名为 `venv` 的虚拟环境，所以无论何时 `cd` 到一个项目中，都会找到相应的虚拟环境。

请注意，在一些操作系统中，你可能需要在上面的命令中使用`python`而不是`python3`。 一些安装规范对Python 2.x版本使用`python`，对3.x版本使用`python3`，而另一些则将`python`映射到3.x版本。

命令执行完成后，当前目录下就会新增一个名为`venv`的目录来存储这个虚拟环境的相关文件。

如果你使用的Python版本低于3.4（包括2.7版本），则不会默认支持虚拟环境。 对于这些版本的Python，在创建虚拟环境之前，需要下载并安装称为virtualenv的第三方工具。 一旦安装了virtualenv，你可以使用以下命令创建一个虚拟环境：

```
$ virtualenv venv
```

不管你用什么方法创建虚拟环境，创建完毕之后还需要激活才能够进入这个虚拟环境。 要激活你的全新虚拟环境，需使用以下命令：

```
$ source venv/bin/activate
(venv) $ _
```

如果你使用的是Microsoft Windows命令提示符窗口，则激活命令稍有不同：

```
$ venv\Scripts\activate
(venv) $ _
```

激活一个虚拟环境，终端会话的环境配置就会被修改，之后你键入`python`的时候，实际上是调用的虚拟环境中的Python解释器。 此外，终端提示符也被修改成包含被激活的虚拟环境的名称的格式。这种激活是临时的和私有的，因此在关闭终端窗口时它们将不会保留，也不会影响其他的会话。 那么，当你需要同时打开多个终端窗口来调试不同的应用时，每个终端窗口都可以激活不同的虚拟环境而不会相互影响。

成功创建和激活了虚拟环境之后，你可以安装Flask了，命令如下：

```
(venv) $ pip install flask
```

想要验证安装是否成功，可以打开Python解释器，并用*import*语句来导入它：

```
>>> import flask
>>> _
```

如果语句没有报错，那么恭喜你，Flask安装成功了！

"Hello, World" Flask应用

Flask网站展示了一个仅有五行代码的简单示例应用程序。 而我会告诉你一个稍微更复杂的例子，它将为你编写更大的应用程序提供一个很好的基础结构。

应用程序是存在于包中的。 在Python中，包含 ***init*** *.py*文件的子目录被视为一个可导入的包。 当你导入一个包时， ***init*** *.py*会执行并定义这个包暴露给外界的属性。

那就创建一个名为 `app` 的包来存放整个应用吧。记得切换到*microblog*目录下，并执行如下命令：

```
(venv) $ mkdir app
```

并在其下创建文件 `__init__.py` ，输入如下的代码：

```
from flask import Flask

app = Flask(__name__)

from app import routes
```

上面的脚本仅仅是从flask中导入的类 `Flask` ，并以此类创建了一个应用程序对象。 传递给 `Flask` 类的 `__name__` 变量是一个Python预定义的变量，它表示当前调用它的模块的名字。 当需要加载相关的资源，如我将在第二章讲到的模板文件，Flask就使用这个位置作为起点来计算绝对路径。 代码的最后，应用程序导入尚未存在的 `routes` 模块。

这段代码，乍一看可能会让人迷惑。

其一，这里有两个实体名为 `app` 。 `app` 包由*app*目录和 ***init*** *.py*脚本来定义构成，并在 `from app import routes` 语句中被引用。 `app` 变量被定义为 ***init*** *.py*脚本中的 `Flask` 类的一个实例，以至于它成为 `app` 包的属性。

其二， `routes` 模块是在底部导入的，而不是在脚本的顶部。 最下面的导入是解决循环导入的问题，这是Flask应用程序的常见问题。 你将会看到 `routes` 模块需要导入在这个脚本中定义的 `app` 变量，因此将 `routes` 的导入放在底部可以避免由于这两个文件之间的相互引用而导致的错误。

那么在 `routes` 模块中有些什么？ 路由是应用程序实现的不同URL。 在Flask中，应用程序路由的处理逻辑被编写为Python函数，称为视图函数。 视图函数被映射到一个或多个路由URL，以便Flask知道当客户端请求给定的URL时执行什么逻辑。

这是需要写入到*app/routes.py*中的第一个视图函数的代码：

```
from app import app

@app.route('/')
@app.route('/index')
def index():
    return "Hello, World!"
```

这个视图函数简单到只返回一个字符串作为问候用语。 函数上面的两个奇怪的`@app.route`行是装饰器，这是Python语言的一个独特功能。 装饰器会修改跟在其后的函数。 装饰器的常见模式是使用它们将函数注册为某些事件的回调函数。 在这种情况下，`@app.route`修饰器在作为参数给出的URL和函数之间创建一个关联。 在这个例子中，有两个装饰器，它们将URL `/` 和`/index`索引关联到这个函数。 这意味着，当Web浏览器请求这两个URL中的任何一个时，Flask将调用该函数并将其返回值作为响应传递回浏览器。这样做是为了在运行这个应用程序的时候会稍微有一点点意义。

要完成应用程序，你需要在定义Flask应用程序实例的顶层（译者注：也就是microblog目录下）创建一个命名为*microblog.py*的Python脚本。 它仅拥有一个导入应用程序实例的行：

```
from app import app
```

还记得两个`app`实体吗？ 在这里，你可以在同一句话中看到两者。 Flask应用程序实例被称为`app`，是`app`包的成员。`from app import app`语句从`app`包导入其成员`app`变量。 如果你觉得这很混乱，你可以重命名包或者变量。

只要确保所做的操作完全正确，那么你就可以看到如下面的项目结构图：

```
microblog/
  venv/
  app/
    __init__.py
    routes.py
  microblog.py
```

不管你信不信，这个应用的第一个版本现在完成了！ 但是在运行之前，需要通过设置`FLASK_APP`环境变量告诉Flask如何导入它：

```
(venv) $ export FLASK_APP=microblog.py
```

如果你使用Microsoft Windows操作系统，在上面的命令中使用`set`替换`export`。

万事俱备，只欠东风！运行如下命令来运行你的第一个Web应用吧：

```
(venv) $ flask run
 * Serving Flask app "microblog"
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

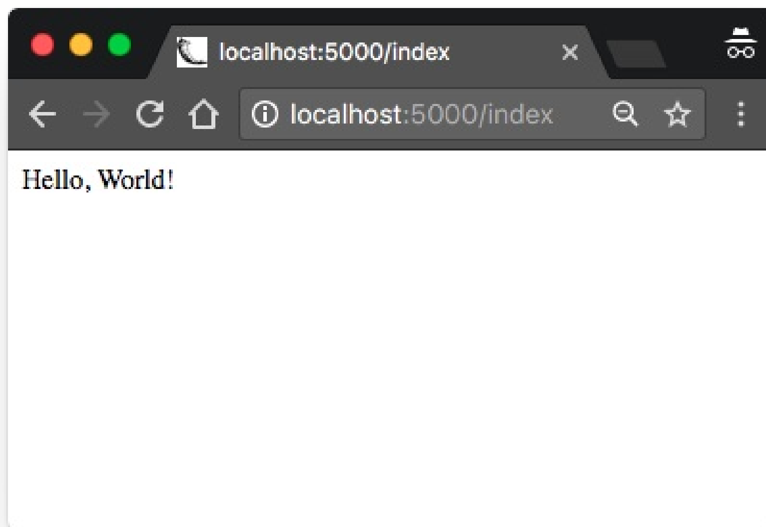
服务启动后将处于阻塞监听状态，将等待客户端连接。 `flask run`的输出表明服务器正在运行在IP地址127.0.0.1上，这是本机的回环IP地址。 这个地址很常见，并有一个更简单的名字，你可能已经看过：*localhost*。 网络服务器监听在指定端口号等待连接。 部署在生产Web服务器上的应用程序通常会在端口443上进行监听，如果不执行加密，则有时会监听80，但启用这些端口需要root权限。 由于此应用程序在开发环境中运行，因此Flask使用自由端口5000。 现在打开您的网络浏览器并在地址栏中输入以下URL：

```
http://localhost:5000/
```

或者，你也可以使用另一个URL：

```
http://localhost:5000/index
```

应用程序路由映射执行了吗？ 第一个URL映射到 `/` ，而第二个映射到 `/ index` 。 这两个路由都与应用程序中唯一的视图函数相关联，所以它们产生相同的输出，即函数返回的字符串。 如果你输入任何其他网址，则会出现错误，因为只有这两个URL被应用程序识别。



完成演示之后，你可以按下Ctrl-C来停止Web服务。

真是可喜可贺！你已经成功地向成为一名Web开发者的道路上迈出了重要的第一步！

本文翻译自 The Flask Mega-Tutorial Part II: Templates

在Flask Mega-Tutorial系列的第二部分中，我将讨论如何使用模板。

学习完第一章之后，你已经拥有了一个虽然简单，但是可以成功运行Web应用，它的文件结构如下：

```
microblog\
  venv\
  app\
    __init__.py
    routes.py
  microblog.py
```

在终端会话中设置环境变量 `FLASK_APP=microblog.py` ，然后执行 `flask run` 命令来运行应用。包含这个应用的Web服务启动之后，你可以通过在Web浏览器的地址栏中键入*http://localhost:5000/* URL来验证。

本章将沿用这个应用，在此之上，你将学习如何生成包含复杂结构和诸多动态组件的网页。如果对这个应用和相关开发流程有所遗忘，请回顾第一章。

本章的*GitHub*链接为：*Browse*, *Zip*, *Diff.*

什么是模板？

我设计的微博应用程序的主页会有一个欢迎用户的标题。虽然目前的应用程序还没有实现用户概念，但这不妨碍我使用一个Python字典来模拟一个用户，如下所示：

```
user = {'username': 'Miguel'}
```

创建模拟对象是一项实用的技术，它可以让你专注于应用程序的一部分，而无需为系统中尚不存在的其他部分分心。 在设计应用程序主页的时候，我可不希望因为没有一个用户系统来分散我的注意力，因此我使用了模拟用户对象，来继续接下来的工作。

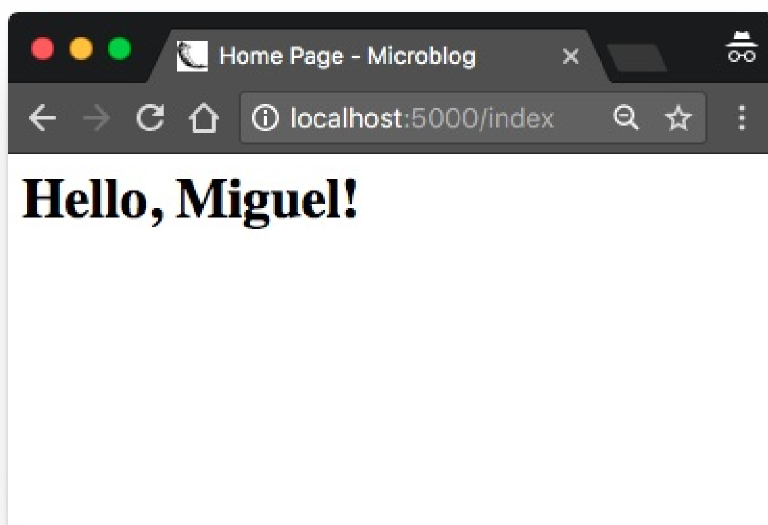
原先的视图函数返回简单的字符串，我现在要将其扩展为包含完整HTML页面元素的字符串，如下所示：

```
from app import app

@app.route('/')
@app.route('/index')
def index():
    user = {'username': 'miguel'}
    return '''
<html>
    <head>
        <title>Home Page - Microblog</title>
    </head>
    <body>
        <h1>Hello, ''' + user['username'] + '''!</h1>
    </body>
</html>'''
```

对HTML标记语言不熟悉的话，建议阅读一下Wikipedia上的简介HTML Markup

利用上述的代码更新这个视图函数，然后再次在浏览器打开它的URL看看结果。



如果我说这个函数返回HTML的方式并不友好的话，你可能会觉得诧异。设想一下，当这个视图函数中的用户和博客不断变化时，里面的代码将会变得多么的复杂。应用的视图函数及其关联的URL也会持续增长。如果哪天我决定更改这个应用的布局，那就不得不更新每个视图函数的HTML字符串。显然，随着应用的扩张，这种方式完全不可行。

将应用程序的后台逻辑和网页布局划分开来，你不觉得更容易组织管理吗？甚至你可以聘请一位Web设计师来设计一个杀手级的网站前端，而你只需要用Python编写后台应用逻辑。

模板有助于实现页面展现和业务逻辑之间的分离。 在Flask中，模板被编写为单独的文件，存储在应用程序包内的*templates*文件夹中。 在确定你在*microblog*目录后，创建一个存储模板的目录：

```
(venv) $ mkdir app/templates
```

在下面可以看到你的第一个模板，它的功能与上面的 `index()` 视图函数返回的HTML页面相似。 把这个文件写在*app/templates/index.html*中：

```
<html>
    <head>
        <title>{{ title }} - Microblog</title>
    </head>
    <body>
        <h1>Hello, {{ user.username }}!</h1>
    </body>
</html>
```

这个HTML页面看起来非常简单，唯一值得关注的地方是 `{{ ... }}` 。 `{{ ... }}` 包含的内容是动态的，只有在运行时才知道具体表示成什么样子。

网页渲染转移到HTML模板之后，视图函数就能被简化：

```
from flask import render_template
from app import app

@app.route('/')
@app.route('/index')
def index():
    user = {'username': 'Miguel'}
    return render_template('index.html', title='Home', user=user)
```

看起来好多了吧？ 赶紧试试这个新版本的应用程序，看看模板是如何工作的。 在浏览器中加载页面后，你需要从浏览器查看HTML源代码并将其与原始模板进行比较。

将模板转换为完整的HTML页面的操作称为渲染。 为了渲染模板，需要从Flask框架中导入一个名为 `render_template()` 的函数。 该函数需要传入模板文件名和模板参数的变量列表，并返回模板中所有占位符都用实际变量值替换后的字符串结果。

`render_template()` 函数调用Flask框架原生依赖的Jinja2模板引擎。 Jinja2用 `render_template()` 函数传入的参数中的相应值替换 `{{...}}` 块。

条件语句

在渲染过程中使用实际值替换占位符，只是Jinja2在模板文件中支持的诸多强大操作之一。 模板也支持在 `{%...%}` 块内使用控制语句。 *index.html*模板的下一个版本添加了一个条件语句：

```
<html>
    <head>
        {% if title %}
        <title>{{ title }} - Microblog</title>
        {% else %}
        <title>Welcome to Microblog!</title>
        {% endif %}
    </head>
    <body>
        <h1>Hello, {{ user.username }}!</h1>
    </body>
</html>
```

现在，模板变得聪明点儿了，如果视图函数忘记给渲染函数传入一个名为`title`的关键字参数，那么模板将显示一个默认的标题，而不是显示一个空的标题。 你可以通过在视图函数的`render_template()`调用中去除`title`参数来试试这个条件语句是如何生效的。

循环

登录后的用户可能想要在主页上查看其他用户的最新动态，针对这个需求，我现在要做的是丰富这个应用来满足它。

我将会故技重施，使用模拟对象的把戏来创建一些模拟用户和动态：

```
from flask import render_template
from app import app

@app.route('/')
@app.route('/index')
def index():
    user = {'username': 'Miguel'}
    posts = [
        {
            'author': {'username': 'John'},
            'body': 'Beautiful day in Portland!'
        },
        {
            'author': {'username': 'Susan'},
            'body': 'The Avengers movie was so cool!'
        }
    ]
    return render_template('index.html', title='Home', user=user, posts=posts)
```

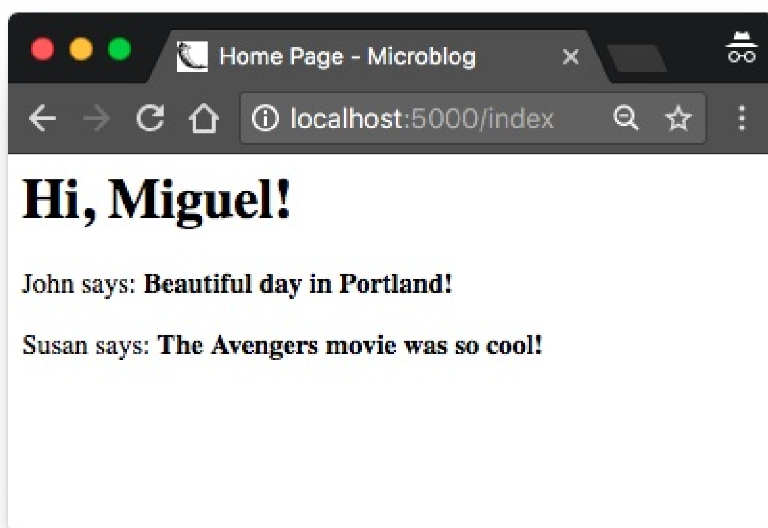
我使用了一个列表来表示用户动态，其中每个元素是一个具有`author`和`body`字段的字典。未来设计用户和其动态时，我将尽可能地保留这些字段名称，以便在使用真实用户和其动态的时候不会出现问题。

在模板方面，我必须解决一个新问题。 用户动态列表拥有的元素数量由视图函数决定。 那么模板不能对有多少个用户动态进行任何假设，因此需要准备好以通用方式渲染任意数量的用户动态。

Jinja2提供了`for`控制结构来应对这类问题：

```
<html>
    <head>
        {% if title %}
        <title>{{ title }} - Microblog</title>
        {% else %}
        <title>Welcome to Microblog</title>
        {% endif %}
    </head>
    <body>
        <h1>Hi, {{ user.username }}!</h1>
        {% for post in posts %}
        <div><p>{{ post.author.username }} says: <b>{{ post.body }}</b></p></div>
        {% endfor %}
    </body>
</html>
```

大道至简，对吧？ 玩玩这个新版本的应用程序，一定要逐步添加更多的内容到用户动态列表，看看模板如何调度以展现视图函数传入的所有用户动态。



模板的继承

绝大多数Web应用程序在页面的顶部都有一个导航栏，其中带有一些常用的链接，例如编辑配置文件，登录，注销等。我可以轻松地用HTML标记语言将导航栏添加到 `index.html` 模板上，但随着应用程序的增长，我将需要在其他页面重复同样的工作。尽量不要编写重复的代码，这是一个良好的编程习惯，毕竟我真的不想在诸多HTML模板上保留同样的代码。

Jinja2有一个模板继承特性，专门解决这个问题。从本质上来讲，就是将所有模板中相同的部分转移到一个基础模板中，然后再从它继承过来。

所以我现在要做的是定义一个名为 `base.html` 的基本模板，其中包含一个简单的导航栏，以及我之前实现的标题逻辑。 您需要在模板文件中*app/templates/base.html*中编写代码如下：

```
<html>
    <head>
      {% if title %}
      <title>{{ title }} - Microblog</title>
      {% else %}
      <title>Welcome to Microblog</title>
      {% endif %}
    </head>
    <body>
        <div>Microblog: <a href="/index">Home</a></div>
        <hr>
        {% block content %}{% endblock %}
    </body>
</html>
```

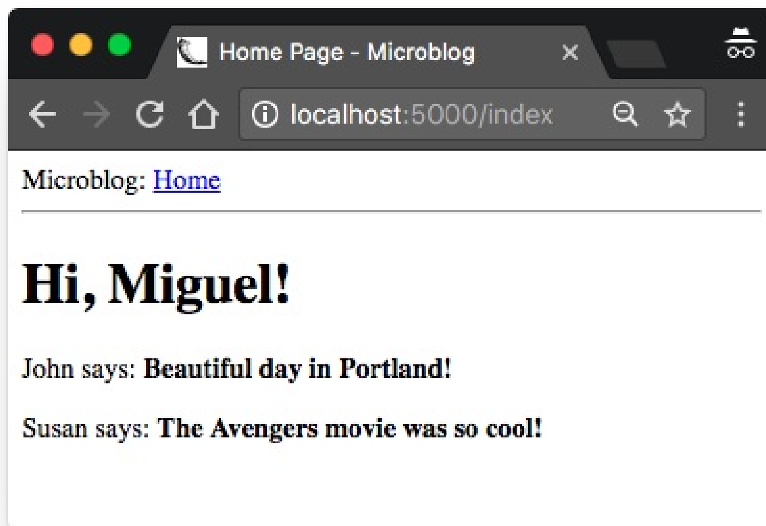
在这个模板中，我使用`block`控制语句来定义派生模板可以插入代码的位置。*block*被赋予一个唯一的名称，派生的模板可以在提供其内容时进行引用。

通过从基础模板*base.html*继承HTML元素，我现在可以简化模板*index.html*了：

```
{% extends "base.html" %}

{% block content %}
    <h1>Hi, {{ user.username }}!</h1>
    {% for post in posts %}
    <div><p>{{ post.author.username }} says: <b>{{ post.body }}</b></p></div>
    {% endfor %}
{% endblock %}
```

自从基础模板*base.html*接手页面的布局之后，我就可以从*index.html*中删除所有这方面的元素，只留下内容部分。`extends`语句用来建立了两个模板之间的继承关系，这样Jinja2才知道当要求呈现`index.html`时，需要将其嵌入到`base.html`中。而两个模板中匹配的`block`语句和其名称`content`，让Jinja2知道如何将这两个模板合并成在一起。现在，扩展应用程序的页面就变得极其方便了，我可以创建从同一个基础模板*base.html*继承的派生模板，这就是我让应用程序的所有页面拥有统一外观布局而不用重复编写代码的秘诀。


Home Page - Microblog
localhost:5000/index
Microblog: Home
Hi, Miguel!
John says: Beautiful day in Portland!
Susan says: The Avengers movie was so cool!

本文翻译自 The Flask Mega-Tutorial Part III: Web Forms

这是Flask Mega-Tutorial系列的第三部分，我将告诉你如何使用*Web*表单。

在第二章中我为应用主页创建了一个简单的模板，并使用诸如用户和用户动态的模拟对象。在本章中，我将解决这个应用程序中仍然存在的众多遗漏之一，那就是如何通过Web表单接受用户的输入。

Web表单是所有Web应用程序中最基本的组成部分之一。 我将使用表单来为用户发表动态和登录认证提供途径。

在继续阅读本章之前，确保你的*microblog*应用程序状态和上一章完结时一致，并且运行时不会报任何错误。

本章的*GitHub*链接为：*Browse*, *Zip*, *Diff.*

Flask-WTF简介

我将使用Flask-WTF插件来处理本应用中的Web表单，它对WTForms进行了浅层次的封装以便和Flask完美结合。这是本应用引入的第一个Flask插件，但绝不是最后一个。插件是Flask生态中的举足轻重的一部分，Flask故意设计为只包含核心功能以保持代码的整洁，并暴露接口以对接解决不同问题的插件。

Flask插件都是常规的Python三方包，可以使用 `pip` 安装。 那就继续在你的虚拟环境中安装Flask-WTF吧：

```
(venv) $ pip install flask-wtf
```

配置

到目前为止，这个应用程序都非常简单，因此我不需要考虑它的配置。 但是，除了最简单的应用，你会发现Flask（也可能是Flask插件）为使用者提供了一些可自由配置的选项。你需要决定传入什么样的配置变量列表到框架中。

有几种途径来为应用指定配置选项。最基本的解决方案是使用 `app.config` 对象，它是一个类似字典的对象，可以将配置以键值的方式存储其中。例如，你可以这样做：

```
app = Flask(__name__)
app.config['SECRET_KEY'] = 'you-will-never-guess'
# ... add more variables here as needed
```

上面的代码虽然可以为应用创建配置，但是我有松耦合的癖好。因此，我不会让配置和应用代码处于同一个部分，而是使用稍微复杂点的结构，将配置保存到一个单独的文件中。

使用类来存储配置变量，才是我真正的风格。我会将这个配置类存储到单独的Python模块，以保持良好的组织结构。下面就让你见识一下这个存储在顶级目录下，名为*config.py*的模块的配置类吧：

```
import os

class Config(object):
    SECRET_KEY = os.environ.get('SECRET_KEY') or 'you-will-never-guess'
```

简单的不像话，有没有？配置设置被定义为 `Config` 类中的属性。一旦应用程序需要更多配置选项，直接依样画葫芦，附加到这个类上即可，稍后如果我发现需要多个配置集，则可以创建它的子类。现在则不用操心。

`SECRET_KEY` 是我添加的唯一配置选项，对大多数Flask应用来说，它都是极其重要的。Flask及其一些扩展使用密钥的值作为加密密钥，用于生成签名或令牌。Flask-WTF插件使用它来保护网页表单免受名为Cross-Site Request Forgery或CSRF（发音为“seasurf”）的恶意攻击。顾名思义，密钥应该是隐密的，因为由它产生的令牌和签名的加密强度保证，取决于除了可信维护者之外，没有任何人能够获得它。

密钥被定义成由 `or` 运算符连接两个项的表达式。第一个项查找环境变量 `SECRET_KEY` 的值，第二个项是一个硬编码的字符串。这种首先检查环境变量中是否存在这个配置，找不到的情况下就使用硬编码字符串的配置变量的模式你将会反复看到。在开发阶段，安全性要求较低，因此可以直接使用硬编码字符串。但是，当应用部署到生产服务器上的时候，我将设置一个独一无二且难以揣摩的环境变量，这样，服务器就拥有了一个别人未知的安全密钥。

拥有了这样一份配置文件，我还需要通知Flask读取并使用它。可以在生成Flask应用之后，利用 `app.config.from_object()` 方法来完成这个操作：

```
from flask import Flask
from config import Config

app = Flask(__name__)
app.config.from_object(Config)

from app import routes
```

导入 `Config` 类的方式，乍一看可能会让人感到困惑，不过如果你注意到从 `flask` 包导入 `Flask` 类的过程，就会发现这其实是类似的操作。显而易见，小写的“config”是Python模块*config.py*的名字，另一个含有大写“C”的是类。

正如我上面提到的，可以使用 `app.config` 中的字典语法来访问配置项。在下面的Python交互式会话中，你可以看到密钥的值：

```
>>> from microblog import app
>>> app.config['SECRET_KEY']
'you-will-never-guess'
```

用户登录表单

Flask-WTF插件使用Python类来表示Web表单。表单类只需将表单的字段定义为类属性即可。

为了再次践行我的松耦合原则，我会将表单类单独存储到名为*app/forms.py*的模块中。就让我们来定义用户登录表单来做一个开始吧，它会要求用户输入username和password，并提供一个“remember me”的复选框和提交按钮：

```
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, BooleanField, SubmitField
from wtforms.validators import DataRequired

class LoginForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired()])
    password = PasswordField('Password', validators=[DataRequired()])
    remember_me = BooleanField('Remember Me')
    submit = SubmitField('Sign In')
```

大多数Flask插件使用`flask_ <name>`命名约定来导入，Flask-WTF的所有内容都在`flask_wtf`包中。在本例中，*app/forms.py*模块的顶部从`flask_wtf`导入了名为`FlaskForm`的基类。

由于Flask-WTF插件本身不提供字段类型，因此我直接从WTForms包中导入了四个表示表单字段的类。每个字段类都接受一个描述或别名作为第一个参数，并生成一个实例来作为`LoginForm`的类属性。

你在一些字段中看到的可选参数`validators`用于验证输入字段是否符合预期。`DataRequired`验证器仅验证字段输入是否为空。更多的验证器将会在未来的表单中接触到。

表单模板

下一步是将表单添加到HTML模板以便渲染到网页上。 令人高兴的是在LoginForm类中定义的字段支持自渲染为HTML元素，所以这个任务相当简单。 我将把登录模板存储在文件*app/templates/login.html* 中，代码如下：

```
{% extends "base.html" %}

{% block content %}
    <h1>Sign In</h1>
    <form action="" method="post">
        {{ form.hidden_tag() }}
        <p>
            {{ form.username.label }}<br>
            {{ form.username(size=32) }}
        </p>
        <p>
            {{ form.password.label }}<br>
            {{ form.password(size=32) }}
        </p>
        <p>{{ form.remember_me() }} {{ form.remember_me.label }}</p>
        <p>{{ form.submit() }}</p>
    </form>
{% endblock %}
```

一如第二章，在这个模板中我再次使用了 `extends` 来继承 `base.html` 基础模板。事实上，我将会对所有的模板继承基础模板，以保持顶部导航栏风格统一。

这个模板需要一个form参数的传入到渲染模板的函数中，form来自于 `LoginForm` 类的实例化，不过我现在还没有编写它。

HTML `<form>` 元素被用作Web表单的容器。 表单的 `action` 属性告诉浏览器在提交用户在表单中输入的信息时应该请求的URL。 当 `action` 设置为空字符串时，表单将被提交给当前地址栏中的URL，即当前页面。 `method` 属性指定了将表单提交给服务器时应该使用的HTTP请求方法。 默认情况下是用 `GET` 请求发送，但几乎在所有情况下，使用 `POST` 请求会提供更好的用户体验，因为这种类型的请求可以在请求的主体中提交表单数据， `GET` 请求将表单字段添加到URL，会使浏览器地址栏变得混乱。

`form.hidden_tag()` 模板参数生成了一个隐藏字段，其中包含一个用于保护表单免受CSRF攻击的 `token` 。 对于保护表单，你需要做的所有事情就是在模板中包括这个隐藏的字段，并在Flask配置中定义 `SECRET_KEY` 变量，Flask-WTF会完成剩下的工作。

如果你以前编写过HTML Web表单，那么你会发现一个奇怪的现象——在此模板中没有HTML表单元素，这是因为表单的字段对象的在渲染时会自动转化为HTML元素。 我只需在需要字段标签的地方加上 `{{ form.<field_name>.label }}` ，需要这个字段的地方加上 `{{ form.<field_name>() }}` 。 对于需要附加HTML属性的字段，可以作为关键字参数传递到函数中。 此模板中的username和password字段将 `size` 作为参数，将其作为属性添加到 `<input>` HTML元素中。 你页也可以通过这种手段为表单字段设置class和id属性。

表单视图

完成这个表单的最后一步就是编写一个新的视图函数来渲染上面创建的模板。

函数的逻辑只需创建一个form实例，并将其传入渲染模板的函数中即可，然后用*/login* URL来关联它。这个视图函数也存储到*app/routes.py*模块中，代码如下：

```
from flask import render_template
from app import app
from app.forms import LoginForm

# ...

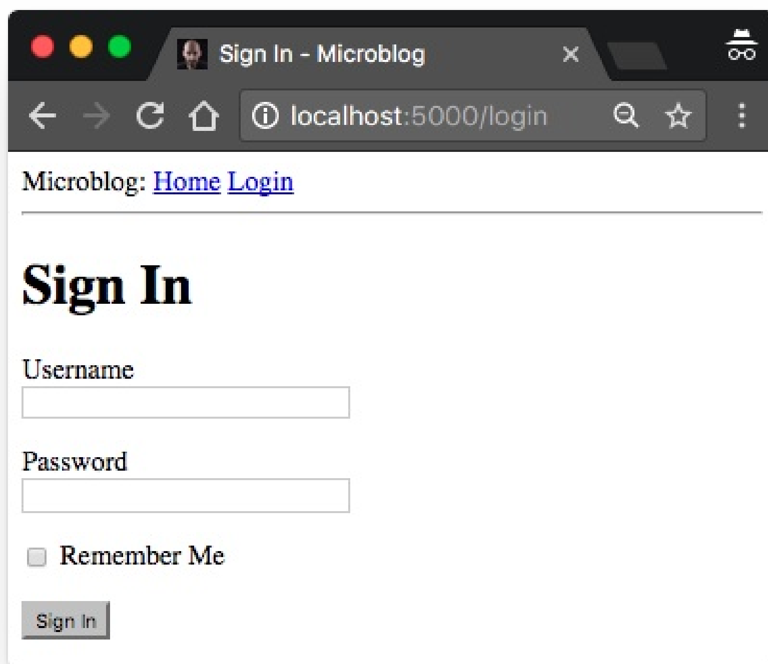
@app.route('/login')
def login():
    form = LoginForm()
    return render_template('login.html', title='Sign In', form=form)
```

我从*forms.py*导入 `LoginForm` 类，并生成了一个实例传入模板。 `form=form` 的语法看起来奇怪，这是Python函数或方法传入关键字参数的方式，左边的 `form` 代表在模板中引用的变量名称，右边则是传入的*form*实例。这就是获取表单字段渲染结果的所有代码了。

在基础模板*templates/base.html*的导航栏上添加登录的链接，以便访问：

```
<div>
    Microblog:
    <a href="/index">Home</a>
    <a href="/login">Login</a>
</div>
```

此时，你可以验证结果了。运行该应用，在浏览器的地址栏中输入 `http://localhost:5000/` ，然后点击顶部导航栏中的“Login”链接来查看新的登录表单。 是不是非常炫酷？



接收表单数据

点击提交按钮，浏览器将显示“Method Not Allowed”错误。为什么呢？ 这是因为之前的登录视图功能到目前为止只完成了一半的工作。 它可以在网页上显示表单，但没有逻辑来处理用户提交的数据。Flask-WTF可以轻松完成这部分工作， 以下是视图函数的更新版本，它接受和验证用户提交的数据：

```
from flask import render_template, flash, redirect

@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        flash('Login requested for user {}, remember_me={}'.format(
            form.username.data, form.remember_me.data))
        return redirect('/index')
    return render_template('login.html', title='Sign In', form=form)
```

这个版本中的第一个新东西是路由装饰器中的 `methods` 参数。 它告诉Flask这个视图函数接受 `GET` 和 `POST` 请求，并覆盖了默认的 `GET` 。 HTTP协议规定对 `GET` 请求需要返回信息给客户端（本例中是浏览器）。 本应用的所有 `GET` 请求都是如此。 当浏览器向服务器提交表单数据时，通常会使用 `POST` 请求（实际上用 `GET` 请求也可以，但这不是推荐的做法）。之前的“Method Not Allowed”错误正是由于视图函数还未配置允许 `POST` 请求。 通过传入 `methods` 参数，你就能告诉Flask哪些请求方法可以被接受。

`form.validate_on_submit()` 实例方法会执行form校验的工作。当浏览器发起 `GET` 请求的时候，它返回 `False` ，这样视图函数就会跳过 `if` 块中的代码，直接转到视图函数的最后一句来渲染模板。

当用户在浏览器点击提交按钮后，浏览器会发送 `POST` 请求。 `form.validate_on_submit()` 就会获取到所有的数据，运行字段各自的验证器，全部通过之后就会返回 `True` ，这表示数据有效。不过，一旦有任意一个字段未通过验证，这个实例方法就会返回 `False` ，引发类似 `GET` 请求那样的表单的渲染并返回给用户。稍后我会在添加代码以实现在验证失败的时候显示一条错误消息。

当 `form.validate_on_submit()` 返回 `True` 时，登录视图函数调用从Flask导入的两个新函数。 `flash()` 函数是向用户显示消息的有效途径。 许多应用使用这个技术来让用户知道某个动作是否成功。我将使用这种机制作为临时解决方案，因为我没有基础架构来真正地登录用户。显示一条消息来确认应用已经收到登录认证凭据，我认为对当前来说已经足够了。

登录视图函数中使用的第二个新函数是 `redirect()` 。这个函数指引浏览器自动重定向到它的参数所关联的URL。当前视图函数使用它将用户重定向到应用的主页。

当你调用 `flash()` 函数后，Flask会存储这个消息，但是却不会奇迹般地直接出现在页面上。模板需要将消息渲染到基础模板中，才能让所有派生出来的模板都能显示出来。更新后的基础模板代码如下：

```
<html>
    <head>
        {% if title %}
        <title>{{ title }} - microblog</title>
        {% else %}
        <title>microblog</title>
        {% endif %}
    </head>
    <body>
        <div>
            Microblog:
            <a href="/index">Home</a>
            <a href="/login">Login</a>
        </div>
        <hr>
        {% with messages = get_flashed_messages() %}
        {% if messages %}
        <ul>
            {% for message in messages %}
            <li>{{ message }}</li>
            {% endfor %}
        </ul>
        {% endif %}
        {% endwith %}
        {% block content %}{% endblock %}
    </body>
</html>
```

此处我用了 `with` 结构在当前模板的上下文中来将 `get_flashed_messages()` 的结果赋值给变量 `messages` 。`get_flashed_messages()` 是Flask中的一个函数，它返回用 `flash()` 注册过的消息列表。接下来的条件结构用来检查变量 `messages` 是否包含元素，如果有，则在 `<ul>` 元素中，为每条消息用 `<li>` 元素来包裹渲染。这种渲染的样式结果看起来不会美观，之后会有主题讲到Web应用的样式。

闪现消息的一个有趣的属性是，一旦通过 `get_flashed_messages` 函数请求了一次，它们就会从消息列表中移除，所以在调用 `flash()` 函数后它们只会出现一次。

时机成熟，再次测试表单吧，将username和password字段留空并点击提交按钮来观察 `DataRequired` 验证器是如何中断提交处理流程的。

完善字段验证

表单字段的验证器可防止无效数据被接收到应用中。 应用处理无效表单输入的方式是重新显示表单，以便用户进行更正。

如果你尝试过提交无效的数据，相信你会注意到，虽然验证机制查无遗漏，却没有给出表单错误的具体线索。下一个任务是通过在验证失败的每个字段旁边添加有意义的错误消息来改善用户体验。

实际上，表单验证器已经生成了这些描述性错误消息，所缺少的不过是模板中的一些额外的逻辑来渲染它们。

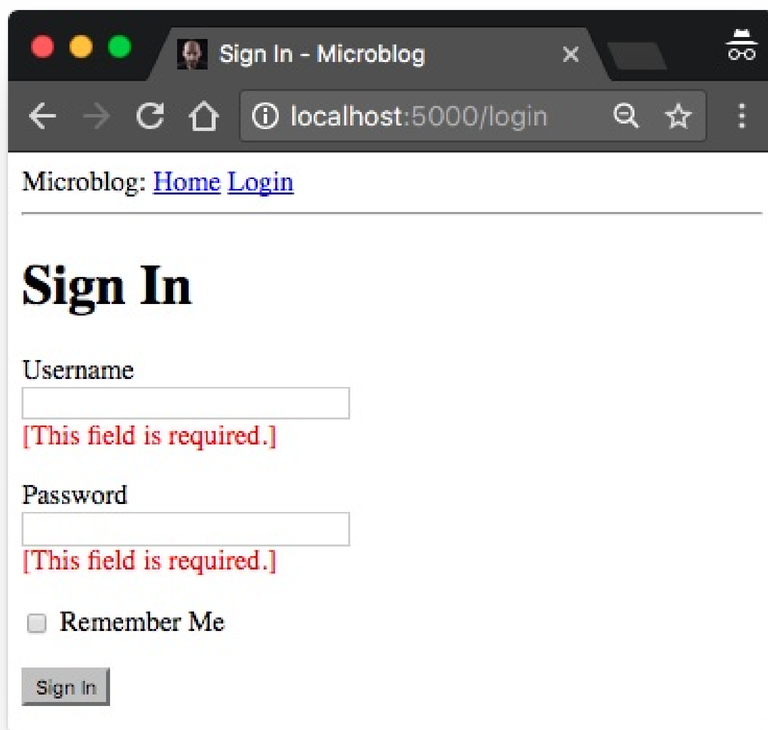
这是给username和password字段添加了验证描述性错误消息渲染逻辑之后的登录模板：

```
{% extends "base.html" %}

{% block content %}
    <h1>Sign In</h1>
    <form action="" method="post">
        {{ form.hidden_tag() }}
        <p>
            {{ form.username.label }}<br>
            {{ form.username(size=32) }}<br>
            {% for error in form.username.errors %}
            <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <p>
            {{ form.password.label }}<br>
            {{ form.password(size=32) }}<br>
            {% for error in form.password.errors %}
            <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <p>{{ form.remember_me() }} {{ form.remember_me.label }}</p>
        <p>{{ form.submit() }}</p>
    </form>
{% endblock %}
```

我做的唯一的改变是，在username和password字段之后添加for循环以便用红色字体来渲染验证器添加的错误信息。通常情况下，拥有验证器的字段都会用 `form.<field_name>.errors` 来渲染错误信息。 一个字段的验证错误信息结果是一个列表，因为字段可以附加多个验证器，并且多个验证器都可能会提供错误消息以显示给用户。

如果你尝试在未填写username和password字段的情况下提交表单，就可以看到显眼的红色错误信息了。



生成链接

现在的登录表单已经相当完整了，但在结束本章之前，我想讨论在模板和重定向中包含链接的妥当方法。 到目前为止，你已经看到了一些定义链接的例子。 例如，这是当前基础模板中的导航栏代码：

```
    <div>
        Microblog:
        <a href="/index">Home</a>
        <a href="/login">Login</a>
    </div>
```

登录视图函数同样定义了一个传入到 `redirect()` 函数作为参数的链接：

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        # ...
        return redirect('/index')
    # ...
```

直接在模板和源文件中硬编码链接存在隐患，如果有一天你决定重新组织链接，那么你将不得不在整个应用中搜索并替换这些链接。

为了更好地管理这些链接，Flask提供了一个名为`url_for()`的函数，它使用URL到视图函数的内部映射关系来生成URL。例如，`url_for('login')`返回`/login`，`url_for('index')`返回`/index`。`url_for()`的参数是*endpoint*名称，也就是视图函数的名字。

你可能会问，为什么使用函数名称而不是URL？事实是，URL比起视图函数名称变更的可能性更高。稍后你会了解到的第二个原因是，一些URL中包含动态组件，手动生成这些URL需要连接多个元素，枯燥乏味且容易出错。`url_for()`生成这种复杂的URL就方便许多。

因此，从现在起，一旦我需要生成应用链接，我就会使用`url_for()`。基础模板中的导航栏部分代码变更如下：

```
    <div>
        Microblog:
        <a href="{{ url_for('index') }}">Home</a>
        <a href="{{ url_for('login') }}">Login</a>
    </div>
```

`login()`视图函数也做了相应变更：

```
from flask import render_template, flash, redirect, url_for

# ...

@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        # ...
        return redirect(url_for('index'))
    # ...
```

本文翻译自 The Flask Mega-Tutorial Part IV: Database

在Flask Mega-Tutorial系列的第四部分，我将告诉你如何使用数据库。

本章的主题是重中之重！大多数应用都需要持久化存储数据，并高效地执行的增删查改的操作，数据库为此而生。

本章的*GitHub*链接为： *Browse*, *Zip*, *Diff.*

Flask中的数据库

Flask本身不支持数据库，相信你已经听说过了。 正如表单那样，这也是Flask有意为之。对使用的数据库插件自由选择，岂不是比被迫适应其中之一，更让人拥有主动权吗？

绝大多数的数据库都提供了Python客户端包，它们之中的大部分都被封装成Flask插件以便更好地和Flask应用结合。数据库被划分为两大类，遵循关系模型的一类是关系数据库，另外的则是非关系数据库，简称*NoSQL*，表现在它们不支持流行的关系查询语言SQL（译者注：部分人也宣称NoSQL代表不仅仅只是SQL）。虽然两类数据库都是伟大的产品，但我认为关系数据库更适合具有结构化数据的应用程序，例如用户列表，用户动态等，而NoSQL数据库往往更适合非结构化数据。 本应用可以像大多数其他应用一样，使用任何一种类型的数据库来实现，但是出于上述原因，我将使用关系数据库。

在第三章中，我向你展示了第一个Flask扩展，在本章中，我还要用到两个。 第一个是Flask-SQLAlchemy，这个插件为流行的SQLAlchemy包做了一层封装以便在Flask中调用更方便，类似*SQLAlchemy*这样的包叫做Object Relational Mapper，简称ORM。 ORM允许应用程序使用高级实体（如类，对象和方法）而不是表和SQL来管理数据库。 ORM的工作就是将高级操作转换成数据库命令。

SQLAlchemy不只是某一款数据库软件的ORM，而是支持包含MySQL、PostgreSQL和SQLite在内的很多数据库软件。简直是太强大了，你可以在开发的时候使用简单易用且无需另起服务的SQLite，需要部署应用到生产服务器上时，则选用更健壮的MySQL或PostgreSQL服务，并且不需要修改应用代码（译者注：只需修改应用配置）。

确认激活虚拟环境之后，利用如下命令来安装Flask-SQLAlchemy插件：

```
(venv) $ pip install flask-sqlalchemy
```

数据库迁移

我所见过的绝大多数数据库教程都是关于如何创建和使用数据库的，却没有指出当需要对现有数据库更新或者添加表结构时，应当如何应对。 这是一项困难的工作，因为关系数据库是以结构化数据为中心的，所以当结构发生变化时，数据库中的已有数据需要被迁移到修改后的结构中。

我将在本章中介绍的第二个插件是Flask-Migrate。 这个插件是Alembic的一个Flask封装，是SQLAlchemy的一个数据库迁移框架。 使用数据库迁移增加了启动数据库时候的一些工作，但这对将来的数据库结构稳健变更来说，是一个很小的代价。

安装Flask-Migrate和安装你见过的其他插件的方式一样：

```
(venv) $ pip install flask-migrate
```

Flask-SQLAlchemy配置

开发阶段，我会使用SQLite数据库，SQLite数据库是开发小型乃至中型应用最方便的选择，因为每个数据库都存储在磁盘上的单个文件中，并且不需要像MySQL和PostgreSQL那样运行数据库服务。

让我们给配置文件添加两个新的配置项：

```
import os
basedir = os.path.abspath(os.path.dirname(__file__))

class Config(object):
    # ...
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL') or \
        'sqlite:///' + os.path.join(basedir, 'app.db')
    SQLALCHEMY_TRACK_MODIFICATIONS = False
```

Flask-SQLAlchemy插件从 `SQLALCHEMY_DATABASE_URI` 配置变量中获取应用的数据库的位置。当回顾第三章可以发现，首先从环境变量获取配置变量，未获取到就使用默认值，这样做是一个好习惯。 本处，我从 `DATABASE_URL` 环境变量中获取数据库URL，如果没有定义，我将其配置为 `basedir` 变量表示的应用顶级目录下的一个名为*app.db*的文件路径。

`SQLALCHEMY_TRACK_MODIFICATIONS` 配置项用于设置数据发生变更之后是否发送信号给应用，我不需要这项功能，因此将其设置为 `False` 。

数据库在应用的表现形式是一个数据库实例，数据库迁移引擎同样如此。它们将会在应用实例化之后进行实例化和注册操作。 `app/__init__.py` 文件变更如下：

```
from flask import Flask
from config import Config
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate

app = Flask(__name__)
app.config.from_object(Config)
db = SQLAlchemy(app)
migrate = Migrate(app, db)

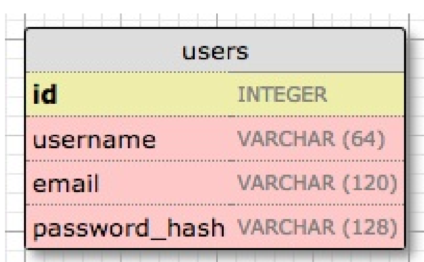
from app import routes, models
```

在这个初始化脚本中我更改了三处。首先，我添加了一个 `db` 对象来表示数据库。然后，我又添加了数据库迁移引擎 `migrate` 。这种注册Flask插件的模式希望你了然于胸，因为大多数Flask插件都是这样初始化的。最后，我在底部导入了一个名为 `models` 的模块，这个模块将会用来定义数据库结构。

数据库模型

定义数据库中一张表及其字段的类，通常叫做数据模型。ORM(SQLAlchemy)会将类的实例关联到数据库表中的数据行，并翻译相关操作。

就让我们从用户模型开始吧，利用 WWW SQL Designer工具，我画了一张图来设计用户表的各个字段（译者注：实际表名为user）：



`id` 字段通常存在于所有模型并用作主键。每个用户都会被数据库分配一个id值，并存储到这个字段中。大多数情况下，主键都是数据库自动赋值的，我只需要提供 `id` 字段作为主键即可。

`username` ， `email` 和 `password_hash` 字段被定义为字符串（数据库术语中的 `VARCHAR` ），并指定其最大长度，以便数据库可以优化空间使用率。 `username` 和 `email` 字段的用途不言而喻， `password_hash` 字段值得提一下。 我想确保我正在构建的应用采用安全最佳实践，因此我不会将用户密码明文存储在数据库中。 明文存储密码的问题是，如果数据库被攻破，攻击者就会获得密码，这对用户隐私来说可能是毁灭性的。 如果使用哈希密码，这就大大提高了安全性。 这将是另一章的主题，所以现在不需分心。

用户表构思完毕之后，我将其用代码实现，并存储到新建的模块*app/models.py*中，代码如下：

```
from app import db

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(64), index=True, unique=True)
    email = db.Column(db.String(120), index=True, unique=True)
    password_hash = db.Column(db.String(128))

    def __repr__(self):
        return '<User {}>'.format(self.username)
```

上面创建的User类继承自db.Model，它是Flask-SQLAlchemy中所有模型的基类。 这个类将表的字段定义为类属性，字段被创建为 `db.Column` 类的实例，它传入字段类型以及其他可选参数，例如，可选参数中允许指示哪些字段是唯一的并且是可索引的，这对高效的数据检索十分重要。

该类的 `__repr__` 方法用于在调试时打印用户实例。在下面的Python交互式会话中你可以看到 `__repr__()` 方法的运行情况：

```
>>> from app.models import User
>>> u = User(username='susan', email='susan@example.com')
>>> u
<User susan>
```

创建数据库迁移存储库

上一节中创建的模型类定义了此应用程序的初始数据库结构（元数据）。 但随着应用的不断增长，很可能会新增、修改或删除数据库结构。 Alembic（Flask-Migrate使用的迁移框架）将以一种不需要重新创建数据库的方式进行数据库结构的变更。

这是一个看起来相当艰巨的任务，为了实现它，Alembic维护一个数据库迁移存储库，它是一个存储迁移脚本的目录。 每当对数据库结构进行更改后，都需要向存储库中添加一个包含更改的详细信息的迁移脚本。 当应用这些迁移脚本到数据库时，它们将按照创建的顺序执行。

Flask-Migrate通过 `flask` 命令暴露来它的子命令。 你已经看过 `flask run` ，这是一个Flask本身的子命令。 Flask-Migrate添加了 `flask db` 子命令来管理与数据库迁移相关的所有事情。 那么让我们通过运行 `flask db init` 来创建microblog的迁移存储库：

```
(venv) $ flask db init
  Creating directory /home/miguel/microblog/migrations ... done
  Creating directory /home/miguel/microblog/migrations/versions ... done
  Generating /home/miguel/microblog/migrations/alembic.ini ... done
  Generating /home/miguel/microblog/migrations/env.py ... done
  Generating /home/miguel/microblog/migrations/README ... done
  Generating /home/miguel/microblog/migrations/script.py.mako ... done
  Please edit configuration/connection/logging settings in
  '/home/miguel/microblog/migrations/alembic.ini' before proceeding.
```

请记住， `flask` 命令依赖于 `FLASK_APP` 环境变量来知道Flask应用入口在哪里。 对于本应用，正如第一章，你需要设置 `FLASK_APP = microblog.py` 。

运行迁移初始化命令之后，你会发现一个名为*migrations*的新目录。该目录中包含一个名为*versions*的子目录以及若干文件。从现在起，这些文件就是你项目的一部分了，应该添加到代码版本管理中去。

第一次数据库迁移

包含映射到`User`数据库模型的用户表的迁移存储库生成后，是时候创建第一次数据库迁移了。 有两种方法来创建数据库迁移：手动或自动。 要自动生成迁移，Alembic会将数据库模型定义的数据库模式与数据库中当前使用的实际数据库模式进行比较。 然后，使用必要的更改来填充迁移脚本，以使数据库模式与应用程序模型匹配。 当前情况是，由于之前没有数据库，自动迁移将把整个User模型添加到迁移脚本中。 `flask db migrate`子命令生成这些自动迁移：

```
(venv) $ flask db migrate -m "users table"
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.autogenerate.compare] Detected added table 'user'
INFO  [alembic.autogenerate.compare] Detected added index 'ix_user_email' on '['email'
]'
INFO  [alembic.autogenerate.compare] Detected added index 'ix_user_username' on '['use
rname']'
  Generating /home/miguel/microblog/migrations/versions/e517276bb1c2_users_table.py ..
. done
```

通过命令输出，你可以了解到Alembic在创建迁移的过程中执行了哪些逻辑。前两行是常规信息，通常可以忽略。 之后的输出表明检测到了一个用户表和两个索引。 然后它会告诉你迁移脚本的输出路径。 `e517276bb1c2`是自动生成的一个用于迁移的唯一标识（你运行的结果会有所不同）。 `-m`可选参数为迁移添加了一个简短的注释。

生成的迁移脚本现在是你项目的一部分了，需要将其合并到源代码管理中。 如果你好奇，并检查了它的代码，就会发现它有两个函数叫`upgrade()`和`downgrade()`。 `upgrade()`函数应用迁移，`downgrade()`函数回滚迁移。 Alembic通过使用降级方法可以将数据库迁移到历史中的任何点，甚至迁移到较旧的版本。

`flask db migrate`命令不会对数据库进行任何更改，只会生成迁移脚本。 要将更改应用到数据库，必须使用`flask db upgrade`命令。

```
(venv) $ flask db upgrade
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade  -> e517276bb1c2, users table
```

因为本应用使用SQLite，所以`upgrade`命令检测到数据库不存在时，会创建它（在这个命令完成之后，你会注意到一个名为*app.db*的文件，即SQLite数据库）。 在使用类似MySQL和PostgreSQL的数据库服务时，必须在运行`upgrade`之前在数据库服务器上创建数据库。

数据库升级和降级流程

目前，本应用还处于初期阶段，但讨论一下未来的数据库迁移战略也无伤大雅。 假设你的开发计算机上存有应用的源代码，并且还将其部署到生产服务器上，运行应用并上线提供服务。

而应用在下一个版本必须对模型进行更改，例如需要添加一个新表。 如果没有迁移机制，这将需要做许多工作。无论是在你的开发机器上，还是在你的服务器上，都需要弄清楚如何变更你的数据库结构才能完成这项任务。

通过数据库迁移机制的支持，在你修改应用中的模型之后，将生成一个新的迁移脚本（`flask db migrate`），你可能会审查它以确保自动生成的正确性，然后将更改应用到你的开发数据库（`flask db upgrade`）。 测试无误后，将迁移脚本添加到源代码管理并提交。

当准备将新版本的应用发布到生产服务器时，你只需要获取包含新增迁移脚本的更新版本的应用，然后运行`flask db upgrade`即可。 Alembic将检测到生产数据库未更新到最新版本，并运行在上一版本之后创建的所有新增迁移脚本。

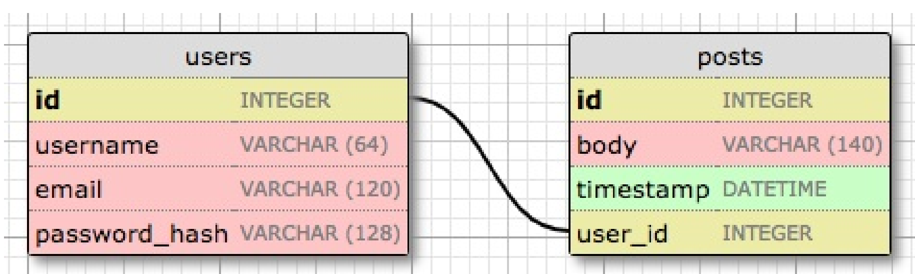
正如我前面提到的，`flask db downgrade`命令可以回滚上次的迁移。 虽然在生产系统上不太可能需要此选项，但在开发过程中可能会发现它非常有用。 你可能已经生成了一个迁移脚本并将其应用，只是发现所做的更改并不完全是你所需要的。 在这种情况下，可以降级数据库，删除迁移脚本，然后生成一个新的来替换它。

数据库关系

关系数据库擅长存储数据项之间的关系。 考虑用户发表动态的情况， 用户将在`user`表中有一个记录，并且这条用户动态将在`post`表中有一个记录。 标记谁写了一个给定的动态的最有效的方法是链接两个相关的记录。

一旦建立了用户和动态之间的关系，数据库就可以在查询中展示它。最小的例子就是当你看一条用户动态的时候需要知道是谁写的。一个更复杂的查询是， 如果你好奇一个用户时，你可能想知道这个用户写的所有动态。 Flask-SQLAlchemy有助于实现这两种查询。

让我们扩展数据库来存储用户动态，以查看实际中的关系。 这是一个新表`post`的设计（译者注：实际表名分别为user和post）：



`post`表将具有必须的`id`、用户动态的`body`和`timestamp`字段。 除了这些预期的字段之外，我还添加了一个`user_id`字段，将该用户动态链接到其作者。 你已经看到所有用户都有一个唯一的`id`主键， 将用户动态链接到其作者的方法是添加对用户`id`的引用，这正是`user_id`字段所在的位置。 这个`user_id`字段被称为外键。 上面的数据库图显示了外键作为该字段和它引用的表的`id`字段之间的链接。 这种关系被称为一对多，因为“一个”用户写了“多”条动态。

修改后的*app/models.py*如下：

```
from datetime import datetime
from app import db

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(64), index=True, unique=True)
    email = db.Column(db.String(120), index=True, unique=True)
    password_hash = db.Column(db.String(128))
    posts = db.relationship('Post', backref='author', lazy='dynamic')

    def __repr__(self):
        return '<User {}>'.format(self.username)

class Post(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    body = db.Column(db.String(140))
    timestamp = db.Column(db.DateTime, index=True, default=datetime.utcnow)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'))

    def __repr__(self):
        return '<Post {}>'.format(self.body)
```

新的“Post”类表示用户发表的动态。 `timestamp` 字段将被编入索引，如果你想按时间顺序检索用户动态，这将非常有用。 我还为其添加了一个 `default` 参数，并传入了 `datetime.utcnow` 函数。 当你将一个函数作为默认值传入后，SQLAlchemy会将该字段设置为调用该函数的值（请注意，在 `utcnow` 之后我没有包含 `()` ，所以我传递函数本身，而不是调用它的结果）。 通常，在服务应用中使用UTC日期和时间是推荐做法。 这可以确保你使用统一的时间戳，无论用户位于何处，这些时间戳会在显示时转换为用户的当地时间。

`user_id` 字段被初始化为 `user.id` 的外键，这意味着它引用了来自用户表的 `id` 值。本处的 `user` 是数据库表的名称，Flask-SQLAlchemy自动设置类名为小写来作为对应表的名称。 `User` 类有一个新的 `posts` 字段，用 `db.relationship` 初始化。这不是实际的数据库字段，而是用户和其动态之间关系的高级视图，因此它不在数据库图表中。对于一对多关系， `db.relationship` 字段通常在“一”的这边定义，并用作访问“多”的便捷方式。因此，如果我有一个用户实例 `u` ，表达式 `u.posts` 将运行一个数据库查询，返回该用户发表过的所有动态。 `db.relationship` 的第一个参数表示代表关系“多”的类。 `backref` 参数定义了代表“多”的类的实例反向调用“一”的时候的属性名称。这将会为用户动态添加一个属性 `post.author` ，调用它将返回给该用户动态的用户实例。 `lazy` 参数定义了这种关系调用的数据库查询是如何执行的，这个我会在后面讨论。不要觉得这些细节没什么意思，本章的结尾将会给出对应的例子。

一旦我变更了应用模型，就需要生成一个新的数据库迁移：

```
(venv) $ flask db migrate -m "posts table"
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.autogenerate.compare] Detected added table 'post'
INFO  [alembic.autogenerate.compare] Detected added index 'ix_post_timestamp' on '['ti
mestamp']'
  Generating /home/miguel/microblog/migrations/versions/780739b227a7_posts_table.py ..
. done
```

并将这个迁移应用到数据库：

```
(venv) $ flask db upgrade
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade e517276bb1c2 -> 780739b227a7, posts
table
```

如果你对项目使用了版本控制，记得将新的迁移脚本添加进去并提交。

表演时刻

经历了一个漫长的过程来定义数据库，我却还没向你展示它们如何使用。 由于应用还没有任何数据库逻辑，所以让我们在Python解释器中来使用以便熟悉它。 立即运行`python`命令来启动Python（在启动解释器之前，确保您的虚拟环境已被激活）。

进入Python交互式环境后，导入数据库实例和模型：

```
>>> from app import db
>>> from app.models import User, Post
```

开始阶段，创建一个新用户：

```
>>> u = User(username='john', email='john@example.com')
>>> db.session.add(u)
>>> db.session.commit()
```

对数据库的更改是在会话的上下文中完成的，你可以通过`db.session`进行访问验证。 允许在会话中累积多个更改，一旦所有更改都被注册，你可以发出一个指令`db.session.commit()`来以原子方式写入所有更改。 如果在会话执行的任何时候出现错误，调用`db.session.rollback()`会中止会话并删除存储在其中的所有更改。 要记住的重要一点是，只有在调用`db.session.commit()`时才会将更改写入数据库。 会话可以保证数据库永远不会处于不一致的状态。

添加另一个用户：

```
>>> u = User(username='susan', email='susan@example.com')
>>> db.session.add(u)
>>> db.session.commit()
```

数据库执行返回所有用户的查询：

```
>>> users = User.query.all()
>>> users
[<User john>, <User susan>]
>>> for u in users:
...     print(u.id, u.username)
...
1 john
2 susan
```

所有模型都有一个`query`属性，它是运行数据库查询的入口。 最基本的查询就是返回该类的所有元素，它被适当地命名为`all()`。 请注意，添加这些用户时，它们的`id`字段依次自动设置为1和2。

另外一种查询方式是，如果你知道用户的`id`，可以用以下方式直接获取用户实例：

```
>>> u = User.query.get(1)
>>> u
<User john>
```

现在添加一条用户动态：

```
>>> u = User.query.get(1)
>>> p = Post(body='my first post!', author=u)
>>> db.session.add(p)
>>> db.session.commit()
```

我不需要为`timestamp`字段设置一个值，因为这个字段有一个默认值，你可以在模型定义中看到。 那么`user_id`字段呢？ 回想一下，我在`User`类中创建的`db.relationship`为用户添加了`posts`属性，并为用户动态添加了`author`属性。 我使用`author`虚拟字段来调用其作者，而不必通过用户ID来处理。 SQLAlchemy在这方面非常出色，因为它提供了对关系和外键的高级抽象。

为了完成演示，让我们看看另外的数据库查询案例：

```
>>> # get all posts written by a user
>>> u = User.query.get(1)
>>> u
<User john>
>>> posts = u.posts.all()
>>> posts
[<Post my first post!>]

>>> # same, but with a user that has no posts
>>> u = User.query.get(2)
>>> u
<User susan>
>>> u.posts.all()
[]

>>> # print post author and body for all posts
>>> posts = Post.query.all()
>>> for p in posts:
...     print(p.id, p.author.username, p.body)
...
1 john my first post!

# get all users in reverse alphabetical order
>>> User.query.order_by(User.username.desc()).all()
[<User susan>, <User john>]
```

Flask-SQLAlchemy文档是学习其对应操作的最好去处。

学完本节内容，我们需要清除这些测试用户和用户动态，以便保持数据整洁和为下一章做好准备：

```
>>> users = User.query.all()
>>> for u in users:
...     db.session.delete(u)
...
>>> posts = Post.query.all()
>>> for p in posts:
...     db.session.delete(p)
...
>>> db.session.commit()
```

Shell上下文

还记得上一节的启动Python解释器之后你做过什么吗？第一件事是运行两条导入语句：

```
>>> from app import db
>>> from app.models import User, Post
```

开发应用时，你经常会在Python shell中测试，所以每次重复上面的导入都会变得枯燥乏味。 `flask shell` 命令是 `flask` 命令集中的另一个非常有用的工具。 `shell` 命令是Flask在继 `run` 之后的实现第二个“核心”命令。 这个命令的目的是在应用的上下文中启动一个Python解释器。 这意味着什么？ 看下面的例子：

```
(venv) $ python
>>> app
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'app' is not defined
>>>

(venv) $ flask shell
>>> app
<Flask 'app'>
```

使用常规的解释器会话时，除非明确地被导入，否则 `app` 对象是未知的，但是当使用 `flask shell` 时，该命令预先导入应用实例。 `flask shell` 的绝妙之处不在于它预先导入了 `app` ，而是你可以配置一个“shell上下文”，也就是可以预先导入一份对象列表。

在*microblog.py*中实现一个函数，它通过添加数据库实例和模型来创建了一个shell上下文环境：

```
from app import app, db
from app.models import User, Post

@app.shell_context_processor
def make_shell_context():
    return {'db': db, 'User': User, 'Post': Post}
```

`app.shell_context_processor` 装饰器将该函数注册为一个shell上下文函数。当 `flask shell` 命令运行时，它会调用这个函数并在shell会话中注册它返回的项目。 函数返回一个字典而不是一个列表，原因是对于每个项目，你必须通过字典的键提供一个名称以便在shell中被调用。

在添加shell上下文处理器函数后，你无需导入就可以使用数据库实例：

```
(venv) $ flask shell
>>> db
<SQLAlchemy engine=sqlite:////Users/migu7781/Documents/dev/flask/microblog2/app.db>
>>> User
<class 'app.models.User'>
>>> Post
<class 'app.models.Post'>
```

本文翻译自The Flask Mega-Tutorial Part V: User Logins

这是Flask Mega-Tutorial系列的第五部分，我将告诉你如何创建一个用户登录子系统。

你在第三章中学会了如何创建用户登录表单，在第四章中学会了运用数据库。本章将教你如何结合这两章的主题来创建一个简单的用户登录系统。

本章的GitHub链接为：*Browse*, *Zip*, *Diff.*

密码哈希

在第四章中，用户模型设置了一个 `password_hash` 字段，到目前为止还没有被使用到。 这个字段的目的是保存用户密码的哈希值，并用于验证用户在登录过程中输入的密码。 密码哈希的实现是一个复杂的话题，应该由安全专家来搞定，不过，已经有数个现成的简单易用且功能完备加密库存在了。

其中一个实现密码哈希的包是Werkzeug，当安装Flask时，你可能会在pip的输出中看到这个包，因为它是Flask的一个核心依赖项。 所以，Werkzeug已经安装在你的虚拟环境中。 以下Python shell会话演示了如何哈希密码：

```
>>> from werkzeug.security import generate_password_hash
>>> hash = generate_password_hash('foobar')
>>> hash
'pbkdf2:sha256:50000$vT9fkZM8$04dfa35c6476acf7e788a1b5b3c35e217c78dc04539d295f011f01f1
8cd2175f'
```

在这个例子中，通过一系列已知没有反向操作的加密操作，将密码 `foobar` 转换成一个长编码字符串，这意味着获得密码哈希值的人将无法使用它逆推出原始密码。 作为一个附加手段，多次哈希相同的密码，你将得到不同的结果，所以这使得无法通过查看它们的哈希值来确定两个用户是否具有相同的密码。

验证过程使用Werkzeug的第二个函数来完成，如下所示：

```
>>> from werkzeug.security import check_password_hash
>>> check_password_hash(hash, 'foobar')
True
>>> check_password_hash(hash, 'barfoo')
False
```

向验证函数传入之前生成的密码哈希值以及用户在登录时输入的密码，如果用户提供的密码执行哈希过程后与存储的哈希值匹配，则返回 `True` ，否则返回 `False` 。

整个密码哈希逻辑可以在用户模型中实现为两个新的方法：

```
from werkzeug.security import generate_password_hash, check_password_hash

# ...

class User(db.Model):
    # ...

    def set_password(self, password):
        self.password_hash = generate_password_hash(password)

    def check_password(self, password):
        return check_password_hash(self.password_hash, password)
```

使用这两种方法，用户对象现在可以在无需持久化存储原始密码的条件下执行安全的密码验证。 以下是这些新方法的示例用法：

```
>>> u = User(username='susan', email='susan@example.com')
>>> u.set_password('mypassword')
>>> u.check_password('anotherpassword')
False
>>> u.check_password('mypassword')
True
```

Flask-Login简介

在本章中，我将向你介绍一个非常受欢迎的Flask插件Flask-Login。 该插件管理用户登录状态，以便用户可以登录到应用，然后用户在导航到该应用的其他页面时，应用会“记得”该用户已经登录。它还提供了“记住我”的功能，允许用户在关闭浏览器窗口后再次访问应用时保持登录状态。可以先在你的虚拟环境中安装Flask-Login来做好准备工作：

```
(venv) $ pip install flask-login
```

和其他插件一样，Flask-Login需要在*app/* ***init*** *py*中的应用实例之后被创建和初始化。 该插件初始化代码如下：

```
# ...
from flask_login import LoginManager

app = Flask(__name__)
# ...
login = LoginManager(app)

# ...
```

为Flask-Login准备用户模型

Flask-Login插件需要在用户模型上实现某些属性和方法。这种做法很棒，因为只要将这些必需项添加到模型中，Flask-Login就没有其他依赖了，它就可以与基于任何数据库系统的用户模型一起工作。

必须的四项如下：

- `is_authenticated`：一个用来表示用户是否通过登录认证的属性，用 `True` 和 `False` 表示。
- `is_active`：如果用户账户是活跃的，那么这个属性是 `True` ，否则就是 `False` （译者注：活跃用户的定义是该用户的登录状态是否通过用户名密码登录，通过“记住我”功能保持登录状态的用户是非活跃的）。
- `is_anonymous`：常规用户的该属性是 `False` ，对特定的匿名用户是 `True` 。
- `get_id()`：返回用户的唯一id的方法，返回值类型是字符串(Python 2下返回unicode字符串).

我可以很容易地实现这四个属性或方法，但是由于它们是相当通用的，因此Flask-Login提供了一个叫做 `UserMixin` 的*mixin*类来将它们归纳其中。 下面演示了如何将mixin类添加到模型中：

```
# ...
from flask_login import UserMixin

class User(UserMixin, db.Model):
    # ...
```

用户加载函数

用户会话是Flask分配给每个连接到应用的用户的存储空间，Flask-Login通过在用户会话中存储其唯一标识符来跟踪登录用户。每当已登录的用户导航到新页面时，Flask-Login将从会话中检索用户的ID，然后将该用户实例加载到内存中。

因为数据库对Flask-Login透明，所以需要应用来辅助加载用户。 基于此，插件期望应用配置一个用户加载函数，可以调用该函数来加载给定ID的用户。 该功能可以添加到*app/models.py*模块中：

```
from app import login
# ...

@login.user_loader
def load_user(id):
    return User.query.get(int(id))
```

使用Flask-Login的 `@login.user_loader` 装饰器来为用户加载功能注册函数。 Flask-Login将字符串类型的参数 `id` 传入用户加载函数，因此使用数字ID的数据库需要如上所示地将字符串转换为整数。

用户登入

让我们回顾一下登录视图函数，它实现了一个模拟登录，只发出一个 `flash()` 消息。 现在，应用可以访问用户数据，并知道如何生成和验证密码哈希值，该视图函数就可以完工了。

```
# ...
from flask_login import current_user, login_user
from app.models import User

# ...

@app.route('/login', methods=['GET', 'POST'])
def login():
    if current_user.is_authenticated:
        return redirect(url_for('index'))
    form = LoginForm()
    if form.validate_on_submit():
        user = User.query.filter_by(username=form.username.data).first()
        if user is None or not user.check_password(form.password.data):
            flash('Invalid username or password')
            return redirect(url_for('login'))
        login_user(user, remember=form.remember_me.data)
        return redirect(url_for('index'))
    return render_template('login.html', title='Sign In', form=form)
```

`login()` 函数中的前两行处理一个非预期的情况：假设用户已经登录，却导航到应用的*/login* URL。 显然这是一个不可能允许的错误场景。 `current_user` 变量来自Flask-Login，可以在处理过程中的任何时候调用以获取用户对象。 这个变量的值可以是数据库中的一个用户对象（Flask-Login通过我上面提供的用户加载函数回调读取），或者如果用户还没有登录，则是一个特殊的匿名用户对象。 还记得那些Flask-Login必须的用户对象属性？ 其中之一是 `is_authenticated`，它可以方便地检查用户是否登录。 当用户已经登录，我只需要重定向到主页。

相比之前的调用 `flash()` 显示消息模拟登录，现在我可以真实地登录用户。 第一步是从数据库加载用户。 利用表单提交的username，我可以查询数据库以找到用户。 为此，我使用了SQLAlchemy查询对象的 `filter_by()` 方法。 `filter_by()` 的结果是一个只包含具有匹配用户名的对象的查询结果集。 因为我知道查询用户的结果只可能是有或者没有，所以我通过调用 `first()` 来完成查询，如果存在则返回用户对象;如果不存在则返回None。 在第四章中，你已经看到当你在查询中调用 `all()` 方法时， 将执行该查询并获得与该查询匹配的所有结果的列表。 当你只需要一个结果时，通常使用 `first()` 方法。

如果使用提供的用户名执行查询并成功匹配，我可以接下来通过调用上面定义的 `check_password()` 方法来检查表单中随附的密码是否有效。 密码验证时，将验证存储在数据库中的密码哈希值与表单中输入的密码的哈希值是否匹配。 所以，现在我有两个可能的错误情况：用户名可能是无效的，或者用户密码是错误的。 在这两种情况下，我都会闪现一条消息，然后重定向到登录页面，以便用户可以再次尝试。

如果用户名和密码都是正确的，那么我调用来自Flask-Login的 `login_user()` 函数。 该函数会将用户登录状态注册为已登录，这意味着用户导航到任何未来的页面时，应用都会将用户实例赋值给 `current_user` 变量。

然后，只需将新登录的用户重定向到主页，我就完成了整个登录过程。

用户登出

提供一个用户登出的途径也是必须的，我将会通过Flask-Login的 `logout_user()` 函数来实现。其视图函数代码如下：

```
# ...
from flask_login import logout_user

# ...

@app.route('/logout')
def logout():
    logout_user()
    return redirect(url_for('index'))
```

为了给用户暴露登出链接，我会在导航栏上实现当用户登录之后，登录链接自动转换成登出链接。修改*base.html*模板的导航栏部分后，代码如下：

```
    <div>
        Microblog:
        <a href="{{ url_for('index') }}">Home</a>
        {% if current_user.is_anonymous %}
        <a href="{{ url_for('login') }}">Login</a>
        {% else %}
        <a href="{{ url_for('logout') }}">Logout</a>
        {% endif %}
    </div>
```

用户实例的 `is_anonymous` 属性是在其模型继承 `UserMixin` 类后Flask-Login添加的，表达式 `current_user.is_anonymous` 仅当用户未登录时的值是 `True` 。

要求用户登录

Flask-Login提供了一个非常有用的功能——强制用户在查看应用的特定页面之前登录。 如果未登录的用户尝试查看受保护的页面，Flask-Login将自动将用户重定向到登录表单，并且只有在登录成功后才重定向到用户想查看的页面。

为了实现这个功能，Flask-Login需要知道哪个视图函数用于处理登录认证。在*app/**init**.py*中添加代码如下：

```
# ...
login = LoginManager(app)
login.login_view = 'login'
```

上面的 `'login'` 值是登录视图函数（endpoint）名，换句话说该名称可用于 `url_for()` 函数的参数并返回对应的URL。

Flask-Login使用名为`@login_required`的装饰器来拒绝匿名用户的访问以保护某个视图函数。当你将此装饰器添加到位于`@app.route`装饰器下面的视图函数上时，该函数将受到保护，不允许未经身份验证的用户访问。以下是该装饰器如何应用于应用的主页视图函数的案例：

```
from flask_login import login_required

@app.route('/')
@app.route('/index')
@login_required
def index():
    # ...
```

剩下的就是实现登录成功之后自定重定向回到用户之前想要访问的页面。当一个没有登录的用户访问被`@login_required`装饰器保护的视图函数时，装饰器将重定向到登录页面，不过，它将在这个重定向中包含一些额外的信息以便登录后的回转。例如，如果用户导航到*/index*，那么`@login_required`装饰器将拦截请求并以重定向到*/login*来响应，但是它会添加一个查询字符串参数来丰富这个URL，如*/login?next=/index*。原始URL设置了`next`查询字符串参数后，应用就可以在登录后使用它来重定向。

下面是一段代码，展示了如何读取和处理`next`查询字符串参数：

```
from flask import request
from werkzeug.urls import url_parse

@app.route('/login', methods=['GET', 'POST'])
def login():
    # ...
    if form.validate_on_submit():
        user = User.query.filter_by(username=form.username.data).first()
        if user is None or not user.check_password(form.password.data):
            flash('Invalid username or password')
            return redirect(url_for('login'))
        login_user(user, remember=form.remember_me.data)
        next_page = request.args.get('next')
        if not next_page or url_parse(next_page).netloc != '':
            next_page = url_for('index')
        return redirect(next_page)
    # ...
```

在用户通过调用Flask-Login的`login_user()`函数登录后，应用获取了`next`查询字符串参数的值。Flask提供一个`request`变量，其中包含客户端随请求发送的所有信息。特别是`request.args`属性，可用友好的字典格式暴露查询字符串的内容。实际上有三种可能的情况需要考虑，以确定成功登录后重定向的位置：

- 如果登录URL中不含`next`参数，那么将会重定向到本应用的主页。
- 如果登录URL中包含`next`参数，其值是一个相对路径（换句话说，该URL不含域名信息），那么将会重定向到本应用的这个相对路径。
- 如果登录URL中包含`next`参数，其值是一个包含域名的完整URL，那么重定向到本应用的主页。

前两种情况很好理解，第三种情况是为了使应用更安全。 攻击者可以在 `next` 参数中插入一个指向恶意站点的URL，因此应用仅在重定向URL是相对路径时才执行重定向，这可确保重定向与应用保持在同一站点中。 为了确定URL是相对的还是绝对的，我使用Werkzeug的 `url_parse()` 函数解析，然后检查 `netloc` 属性是否被设置。

在模板中显示已登录的用户

你还记得在实现用户子系统之前的第二章中，我创建了一个模拟的用户来帮助我设计主页的事情吗？ 现在，应用实现了真正的用户，我就可以删除模拟用户了。 取而代之，我会在模板中使用Flask-Login的 `current_user` ：

```
{% extends "base.html" %}

{% block content %}
    <h1>Hi, {{ current_user.username }}!</h1>
    {% for post in posts %}
    <div><p>{{ post.author.username }} says: <b>{{ post.body }}</b></p></div>
    {% endfor %}
{% endblock %}
```

并且我可以在视图函数传入渲染模板函数的参数中删除 `user` 了：

```
@app.route('/')
@app.route('/index')
def index():
    # ...
    return render_template("index.html", title='Home Page', posts=posts)
```

这正是测试登录和注销功能运作机制的好时机。 由于仍然没有用户注册功能，所以添加用户到数据库的唯一方法是通过Python shell执行，所以运行 `flask shell` 并输入以下命令来注册用户：

```
>>> u = User(username='susan', email='susan@example.com')
>>> u.set_password('cat')
>>> db.session.add(u)
>>> db.session.commit()
```

如果启动应用并尝试访问*http://localhost:5000/*或*http://localhost:5000/index*，会立即重定向到登录页面。在使用之前添加到数据库的凭据登录后，就会跳转回到之前访问的页面，并看到其中的个性化欢迎。

用户注册

本章要构建的最后一项功能是注册表单，以便用户可以通过Web表单进行注册。 让我们在*app/forms.py*中创建Web表单类来开始吧：

```
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, BooleanField, SubmitField
from wtforms.validators import ValidationError, DataRequired, Email, EqualTo
from app.models import User

# ...

class RegistrationForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired()])
    email = StringField('Email', validators=[DataRequired(), Email()])
    password = PasswordField('Password', validators=[DataRequired()])
    password2 = PasswordField(
        'Repeat Password', validators=[DataRequired(), EqualTo('password')])
    submit = SubmitField('Register')

    def validate_username(self, username):
        user = User.query.filter_by(username=username.data).first()
        if user is not None:
            raise ValidationError('Please use a different username.')

    def validate_email(self, email):
        user = User.query.filter_by(email=email.data).first()
        if user is not None:
            raise ValidationError('Please use a different email address.')
```

代码中与验证相关的几处相当有趣。首先，对于 `email` 字段，我在 `DataRequired` 之后添加了第二个验证器，名为 `Email` 。 这个来自WTForms的另一个验证器将确保用户在此字段中键入的内容与电子邮件地址的结构相匹配。

由于这是一个注册表单，习惯上要求用户输入密码两次，以减少输入错误的风险。 出于这个原因，我提供了 `password` 和 `password2` 字段。 第二个password字段使用另一个名为 `EqualTo` 的验证器，它将确保其值与第一个password字段的值相同。

我还为这个类添加了两个方法，名为 `validate_username()` 和 `validate_email()` 。 当添加任何匹配模式 `validate_ <field_name>` 的方法时，WTForms将这些方法作为自定义验证器，并在已设置验证器之后调用它们。 本处，我想确保用户输入的username和email不会与数据库中已存在的数据冲突，所以这两个方法执行数据库查询，并期望结果集为空。 否则，则通过 `ValidationError` 触发验证错误。 异常中作为参数的消息将会在对应字段旁边显示，以供用户查看。

我需要一个HTML模板以便在网页上显示这个表单，我其存储在*app/templates/register.html*文件中。 这个模板的构造与登录表单类似：

```
{% extends "base.html" %}

{% block content %}
    <h1>Register</h1>
    <form action="" method="post">
        {{ form.hidden_tag() }}
        <p>
            {{ form.username.label }}<br>
            {{ form.username(size=32) }}<br>
            {% for error in form.username.errors %}
            <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <p>
            {{ form.email.label }}<br>
            {{ form.email(size=64) }}<br>
            {% for error in form.email.errors %}
            <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <p>
            {{ form.password.label }}<br>
            {{ form.password(size=32) }}<br>
            {% for error in form.password.errors %}
            <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <p>
            {{ form.password2.label }}<br>
            {{ form.password2(size=32) }}<br>
            {% for error in form.password2.errors %}
            <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <p>{{ form.submit() }}</p>
    </form>
{% endblock %}
```

登录表单模板需要在其表单之下添加一个链接来将未注册的用户引导到注册页面：

```
    <p>New User? <a href="{{ url_for('register') }}">Click to Register!</a></p>
```

最后，我来实现处理用户注册的视图函数，存储在*app/routes.py*中，代码如下：

```
from app import db
from app.forms import RegistrationForm

# ...

@app.route('/register', methods=['GET', 'POST'])
def register():
    if current_user.is_authenticated:
        return redirect(url_for('index'))
    form = RegistrationForm()
    if form.validate_on_submit():
        user = User(username=form.username.data, email=form.email.data)
        user.set_password(form.password.data)
        db.session.add(user)
        db.session.commit()
        flash('Congratulations, you are now a registered user!')
        return redirect(url_for('login'))
    return render_template('register.html', title='Register', form=form)
```

这个视图函数的逻辑也是一目了然，我首先确保调用这个路由的用户没有登录。表单的处理方式和登录的方式一样。在 `if validate_on_submit()` 条件块下，完成的逻辑如下：使用获取自表单的username、email和password创建一个新用户，将其写入数据库，然后重定向到登录页面以便用户登录。

精雕细琢之后，用户已经能够在此应用上注册帐户，并进行登录和注销。 请确保你尝试了我在注册表单中添加的所有验证功能，以便更好地了解其工作原理。 我将在未来的章节中再次更新用户认证子系统，以增加额外的功能，比如允许用户在忘记密码的情况下重置密码。 不过对于目前的应用来讲，这已经无碍于继续构建了。

本文翻译自 The Flask Mega-Tutorial Part VI: Profile Page and Avatars

这是Flask Mega-Tutorial系列的第六部分，我将告诉你如何创建个人主页。

本章将致力于为应用添加个人主页。个人主页用来展示用户的相关信息，其个人信息由本人录入。 我将为你展示如何动态地生成每个用户的主页，并提供一个编辑页面给他们来更新个人信息。

本章的GitHub链接为：*Browse*, *Zip*, *Diff.*

个人主页

作为创建个人主页的第一步，让我们为其URL */user/* 新建一个对应的视图函数。

```
@app.route('/user/<username>')
@login_required
def user(username):
    user = User.query.filter_by(username=username).first_or_404()
    posts = [
        {'author': user, 'body': 'Test post #1'},
        {'author': user, 'body': 'Test post #2'}
    ]
    return render_template('user.html', user=user, posts=posts)
```

我用来装饰该视图函数的 `@app.route` 装饰器看起来和之前的有点不一样。 本例中被 `<` 和 `>` 包裹的URL `<username>` 是动态的。 当一个路由包含动态组件时，Flask将接受该部分URL中的任何文本，并将以实际文本作为参数调用该视图函数。 例如，如果客户端浏览器请求URL `/user/susan`，则视图函数将被调用，其参数 `username` 被设置为 `'susan'`。 因为这个视图函数只能被已登录的用户访问，所以我添加了 `@login_required` 装饰器。

这个视图函数的实现相当简单。 我首先会尝试在数据库中以用户名来查询和加载用户。 之前你见过通过调用 `all()` 来得到所有的结果的查询，或是调用 `first()` 来得到结果中的第一个或者结果集为空时返回 `None` 的查询。 在本视图函数中，我使用了 `first()` 的变种方法，名为 `first_or_404()`，当有结果时它的工作方式与 `first()` 完全相同，但是在没有结果的情况下会自动发送404 error给客户端。 以这种方式执行查询，我省去检查用户是否返回的步骤，因为当用户名不存在于数据库中时，函数将不会返回，而是会引发404异常。

如果执行数据库查询没有触发404错误，那么这意味着找到了具有给定用户名的用户。 接下来，我为这个用户初始化一个虚拟的用户动态列表，最后用传入的用户对象和用户动态列表渲染一个新的*user.html*模板。

*user.html*模板如下所示：

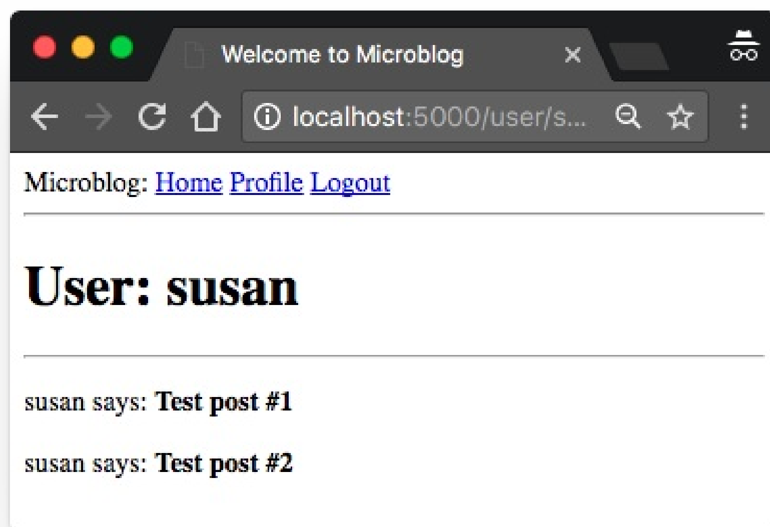
```
{% extends "base.html" %}

{% block content %}
    <h1>User: {{ user.username }}</h1>
    <hr>
    {% for post in posts %}
    <p>
    {{ post.author.username }} says: <b>{{ post.body }}</b>
    </p>
    {% endfor %}
{% endblock %}
```

个人主页虽然已经完成了，但是网站上却没有一个入口链接。我将会在顶部的导航栏中添加这个入口链接，以便用户可以轻松查看自己的个人资料：

```
    <div>
      Microblog:
      <a href="{{ url_for('index') }}">Home</a>
      {% if current_user.is_anonymous %}
      <a href="{{ url_for('login') }}">Login</a>
      {% else %}
      <a href="{{ url_for('user', username=current_user.username) }}">Profile</a>
      <a href="{{ url_for('logout') }}">Logout</a>
      {% endif %}
    </div>
```

这里唯一有趣的变化是用来生成链接到个人主页的 `url_for()` 调用。 由于个人主页视图函数接受一个动态参数，所以 `url_for()` 函数接收一个值作为关键字参数。 由于这是一个指向当前登录个人主页的链接，我可以使用Flask-Login的 `current_user` 对象来生成正确的URL。



尝试点击顶部的 `Profile` 链接就能将你带到自己的个人主页。 此时，虽然没有链接来访问其他用户的主页，但是如果要访问这些页面，则可以在浏览器的地址栏中手动输入网址。 例如，如果你在应用中注册了名为“john”的用户，则可以通过在地址栏中键入*http://*

*localhost:5000/user/john*来查看该用户的个人主页。

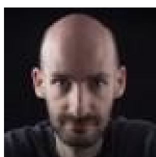
头像

我相信你也觉得我刚刚建立的个人主页非常枯燥乏味。为了使它们更加有趣，我将添加用户头像。与其在服务器上处理大量的上传图片，我将使用Gravatar为所有用户提供图片服务。

Gravatar服务使用起来非常简单。 要请求给定用户的图片，使用格式为*https://www.gravatar.com/avatar/*的URL即可，其中`<hash>`是用户的电子邮件地址的MD5哈希值。 在下面，你可以看到如何生成电子邮件为`john@example.com`的用户的Gravatar URL：

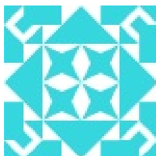
```
>>> from hashlib import md5
>>> 'https://www.gravatar.com/avatar/' + md5(b'john@example.com').hexdigest()
'https://www.gravatar.com/avatar/d4c74594d841139328695756648b6bd6'
```

如果你想看一个实际的例子，我自己的Gravatar URL是*https://www.gravatar.com/avatar/729e26a2a2c7ff24a71958d4aa4e5f35*。Gravatar返回的图片如下：



默认情况下，返回的图像大小是80x80像素，但可以通过向URL的查询字符串添加`s`参数来请求不同大小的图片。 例如，要获得我自己128x128像素的头像，该URL是*https://www.gravatar.com/avatar/729e26a2a2c7ff24a71958d4aa4e5f35?s=128*。

另一个可传递给Gravatar的有趣参数是`d`，它让Gravatar为没有向服务注册头像的用户提供的随机头像。 我最喜欢的随机头像类型是“identicon”，它为每个邮箱都返回一个漂亮且不重复的几何设计图片。 如下：



请注意，一些Web浏览器插件（如Ghostery）会屏蔽Gravatar图像，因为它们认为Automattic（Gravatar服务的所有者）可以根据你发送的获取头像的请求来判断你正在访问的网站。 如果在浏览器中看不到头像，你在排查问题的时候可以考虑以下是否在浏览器中安装了此类插件。

由于头像与用户相关联，所以将生成头像URL的逻辑添加到用户模型是有道理的。

```
from hashlib import md5
# ...

class User(UserMixin, db.Model):
    # ...
    def avatar(self, size):
        digest = md5(self.email.lower().encode('utf-8')).hexdigest()
        return 'https://www.gravatar.com/avatar/{}?d=identicon&s={}'.format(
            digest, size)
```

`User` 类新增的 `avatar()` 方法需要传入需求头像的像素大小，并返回用户头像图片的URL。对于没有注册头像的用户，将生成“identicon”类的随机图片。 为了生成MD5哈希值，我首先将电子邮件转换为小写，因为这是Gravatar服务所要求的。 然后，因为Python中的MD5的参数类型需要是字节而不是字符串，所以在将字符串传递给该函数之前，需要将字符串编码为字节。

如果你对Gravatar服务很有兴趣，可以学习他们的文档。

下一步需要将头像图片插入到个人主页的模板中：

```
{% extends "base.html" %}

{% block content %}
    <table>
        <tr valign="top">
            <td><img src="{{ user.avatar(128) }}"></td>
            <td><h1>User: {{ user.username }}</h1></td>
        </tr>
    </table>
    <hr>
    {% for post in posts %}
    <p>
    {{ post.author.username }} says: <b>{{ post.body }}</b>
    </p>
    {% endfor %}
{% endblock %}
```

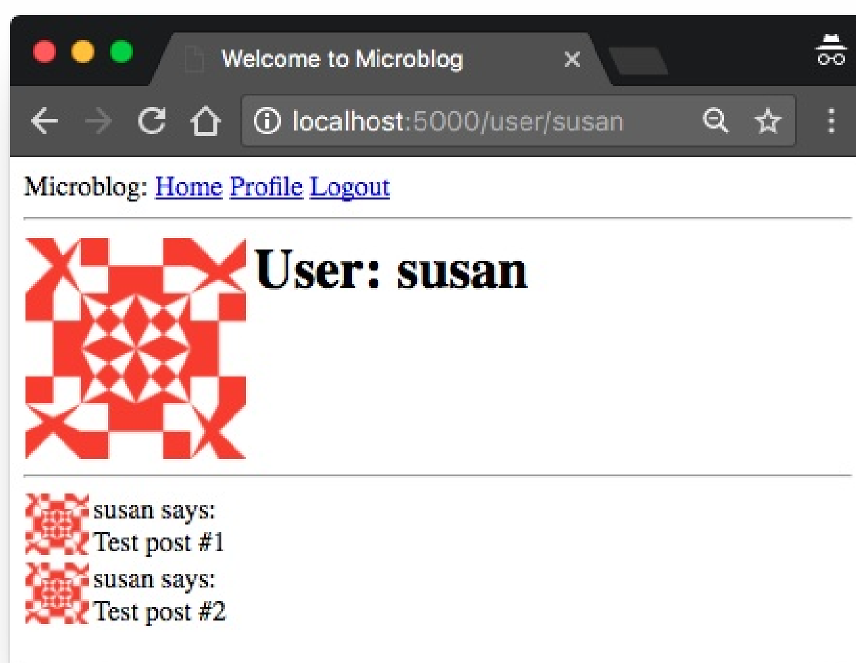
使用 `User` 类来返回头像URL的好处是，如果有一天我不想继续使用Gravatar头像了，我可以重写 `avatar()` 方法来返回其他头像服务网站的URL，所有的模板将自动显示新的头像。

我的个人主页的顶部有一个不错的大头像，不止如此，底下的所有用户动态都会有一个小头像。 对于个人主页而言，所有的头像当然都是对应用户的。我将会在主页面上实现每个用户动态都用其作者的头像来装饰，这样一来看起来就非常棒了。

为了显示用户动态的头像，我只需要在模板中进行一个小小的更改：

```
{% extends "base.html" %}

{% block content %}
    <table>
        <tr valign="top">
            <td><img src="{{ user.avatar(128) }}"></td>
            <td><h1>User: {{ user.username }}</h1></td>
        </tr>
    </table>
    <hr>
    {% for post in posts %}
    <table>
        <tr valign="top">
            <td><img src="{{ post.author.avatar(36) }}"></td>
            <td>{{ post.author.username }} says:<br>{{ post.body }}</td>
        </tr>
    </table>
    {% endfor %}
{% endblock %}
```



使用Jinja2子模板

我设计的个人主页，使用头像和文字组合的方式来展示了用户动态。 现在我想在主页也使用类似的风格来布局。 我可以复制/粘贴来处理用户动态渲染的模板部分，但这实际上并不理想，因为之后如果我想要对此布局进行更改，我将不得不记住要更新两个模板。

取而代之，我要创建一个只渲染一条用户动态的子模板，然后在*user.html*和*index.html*模板中引用它。 首先，我要创建这个只有一条用户动态HTML元素的子模板。 我将其命名为*app/templates/_post.html*， `_` 前缀只是一个命名约定，可以帮助我识别哪些模板文件是子

模板。

```
    <table>
        <tr valign="top">
            <td><img src="{{ post.author.avatar(36) }}"></td>
            <td>{{ post.author.username }} says:<br>{{ post.body }}</td>
        </tr>
    </table>
```

我在*user.html*模板中使用了Jinja2的`include`语句来调用该子模板：

```
{% extends "base.html" %}

{% block content %}
    <table>
        <tr valign="top">
            <td><img src="{{ user.avatar(128) }}"></td>
            <td><h1>User: {{ user.username }}</h1></td>
        </tr>
    </table>
    <hr>
    {% for post in posts %}
        {% include '_post.html' %}
    {% endfor %}
{% endblock %}
```

应用的主页还没有完善，所以现在我不打算在其中添加这个功能。

更多有趣的个人资料

新增的个人主页存在的一个问题是，真正显示的内容不够丰富。 用户喜欢在个人主页上展示他们的相关信息，所以我会让他们写一些自我介绍并在这里展示。 我也将跟踪每个用户最后一次访问该网站的时间，并显示在他们的个人主页上。

为了支持所有这些额外的信息，首先需要做的是用两个新的字段扩展数据库中的用户表：

```
class User(UserMixin, db.Model):
    # ...
    about_me = db.Column(db.String(140))
    last_seen = db.Column(db.DateTime, default=datetime.utcnow)
```

每次数据库被修改时，都需要生成数据库迁移。 在第四章中，我向你展示了如何设置应用以通过迁移脚本跟踪数据库的变更。 现在有两个新的字段我想添加到数据库中，所以第一步是生成迁移脚本：

```
(venv) $ flask db migrate -m "new fields in user model"
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.autogenerate.compare] Detected added column 'user.about_me'
INFO  [alembic.autogenerate.compare] Detected added column 'user.last_seen'
  Generating /home/miguel/microblog/migrations/versions/37f06a334dbf_new_fields_in_use
r_model.py ... done
```

`migrate` 命令的输出表示一切正确运行，因为它显示 `User` 类中的两个新字段已被检测到。现在我可以将此更改应用于数据库：

```
(venv) $ flask db upgrade
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade 780739b227a7 -> 37f06a334dbf, new fi
elds in user model
```

我希望你认识到使用迁移框架是多么有用。 数据库中的用户数据仍然存在，迁移框架如同实施手术教学般地精准执行迁移脚本中的更改并且不损坏任何数据。

下一步，我将会把新增的两个字段增加到个人主页中：

```
{% extends "base.html" %}

{% block content %}
    <table>
        <tr valign="top">
            <td><img src="{{ user.avatar(128) }}"></td>
            <td>
                <h1>User: {{ user.username }}</h1>
                {% if user.about_me %}<p>{{ user.about_me }}</p>{% endif %}
                {% if user.last_seen %}<p>Last seen on: {{ user.last_seen }}</p>{% end
if %}
            </td>
        </tr>
    </table>
    ...
{% endblock %}
```

请注意，我用Jinja2的条件语句来封装了这两个字段，因为我只希望它们在设置后才可见。 目前，所有用户的这两个字段都是空的，所以如果现在运行应用，则不会看到这些字段。

记录用户的最后访问时间

让我们从更容易实现的 `last_seen` 字段开始。 我想要做的就是一旦某个用户向服务器发送请求，就将当前时间写入到这个字段。

为每个视图函数添加更新这个字段的逻辑，这么做非常的枯燥乏味。在视图函数处理请求之前执行一段简单的代码逻辑在Web应用中十分常见，因此Flask提供了一个内置功能来实现它。解决方案如下：

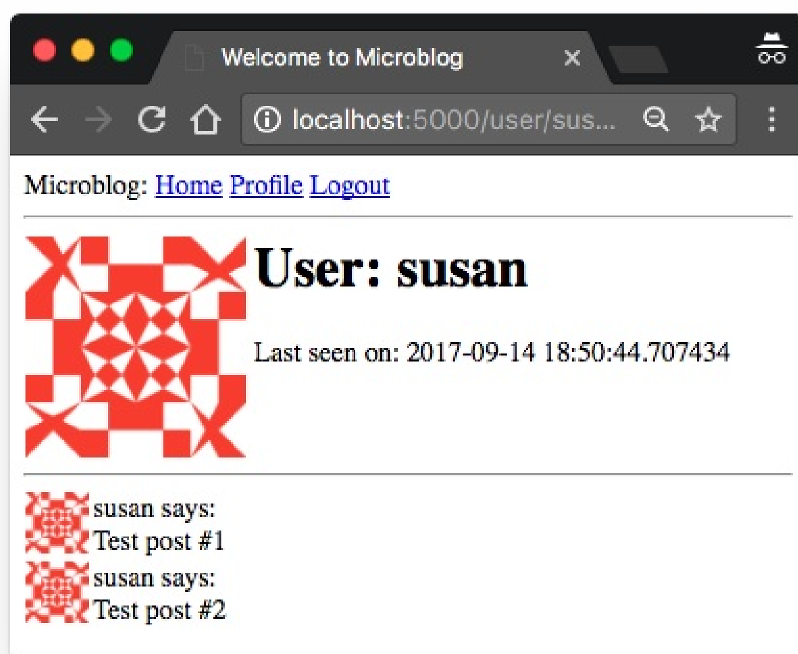
```
from datetime import datetime

@app.before_request
def before_request():
    if current_user.is_authenticated:
        current_user.last_seen = datetime.utcnow()
        db.session.commit()
```

Flask中的 `@before_request` 装饰器注册在视图函数之前执行的函数。这是非常有用的，因为现在我可以在一处地方编写代码，并让它在任何视图函数之前被执行。该代码简单地实现了检查 `current_user` 是否已经登录，并在已登录的情况下将 `last_seen` 字段设置为当前时间。我之前提到过，应用应该以一致的时间单位工作，标准做法是使用UTC时区，使用系统的本地时间不是一个好主意，因为如果那么的话，数据库中存储的时间取决于你的时区。最后一步是提交数据库会话，以便将上面所做的更改写入数据库。如果你想知道为什么在提交之前没有 `db.session.add()` ，考虑在引用 `current_user` 时，Flask-Login将调用用户加载函数，该函数将运行一个数据库查询并将目标用户添加到数据库会话中。所以你可以在这个函数中再次添加用户，但是这不是必须的，因为它已经在那里了。

如果在进行此更改后查看你的个人主页，则会看到“Last seen on”行，并且时间非常接近当前时间。 如果你离开个人主页，然后返回，你会看到时间在不断更新。

事实上，我在存储时间和在个人主页显示时间的时候，使用的都是UTC时区。 除此之外，显示的时间格式也可能不是你所预期的，因为实际上它是Python datetime对象的内部表示。 现在，我不会操心这两个问题，因为我将在后面的章节中讨论在Web应用中处理日期和时间的主题。



个人资料编辑器

我还需要给用户一个表单，让他们输入一些个人资料。 表单将允许用户更改他们的用户名，并且写一些个人介绍，以存储在新的 `about_me` 字段中。 让我们开始为它写一个表单类吧：

```
from wtforms import StringField, TextAreaField, SubmitField
from wtforms.validators import DataRequired, Length

# ...

class EditProfileForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired()])
    about_me = TextAreaField('About me', validators=[Length(min=0, max=140)])
    submit = SubmitField('Submit')
```

我在这个表单中使用了一个新的字段类型和一个新的验证器。 对于“about_me”字段，我使用 `TextAreaField` ，这是一个多行输入文本框，用户可以在其中输入文本。 为了验证这个字段的长度，我使用了 `Length` ，它将确保输入的文本在0到140个字符之间，因为这是我为数据库中的相应字段分配的空间。

该表单的渲染模板代码如下：

```
{% extends "base.html" %}

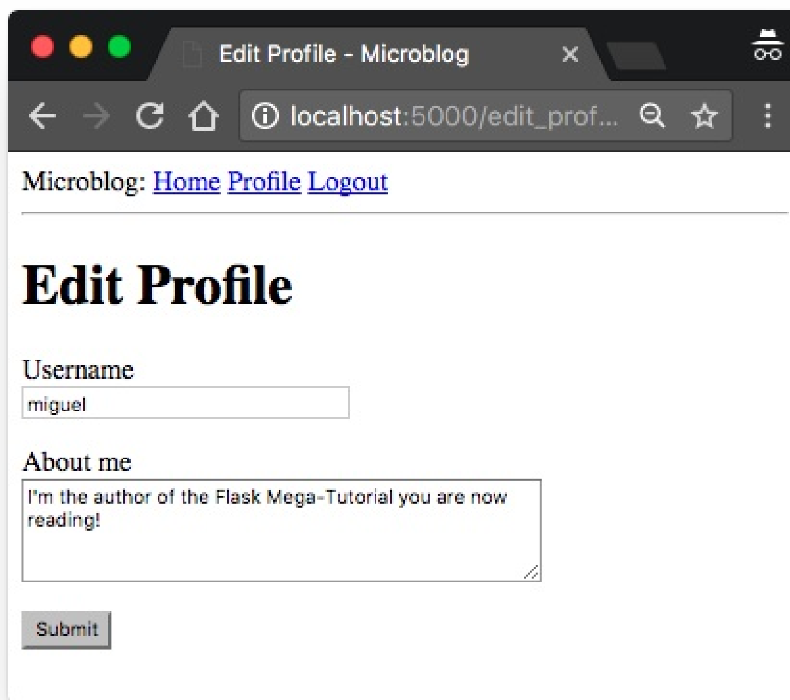
{% block content %}
    <h1>Edit Profile</h1>
    <form action="" method="post">
        {{ form.hidden_tag() }}
        <p>
            {{ form.username.label }}<br>
            {{ form.username(size=32) }}<br>
            {% for error in form.username.errors %}
            <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <p>
            {{ form.about_me.label }}<br>
            {{ form.about_me(cols=50, rows=4) }}<br>
            {% for error in form.about_me.errors %}
            <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <p>{{ form.submit() }}</p>
    </form>
{% endblock %}
```

最后一步，使用视图函数将它们结合起来：

```
from app.forms import EditProfileForm

@app.route('/edit_profile', methods=['GET', 'POST'])
@login_required
def edit_profile():
    form = EditProfileForm()
    if form.validate_on_submit():
        current_user.username = form.username.data
        current_user.about_me = form.about_me.data
        db.session.commit()
        flash('Your changes have been saved.')
        return redirect(url_for('edit_profile'))
    elif request.method == 'GET':
        form.username.data = current_user.username
        form.about_me.data = current_user.about_me
    return render_template('edit_profile.html', title='Edit Profile',
                           form=form)
```

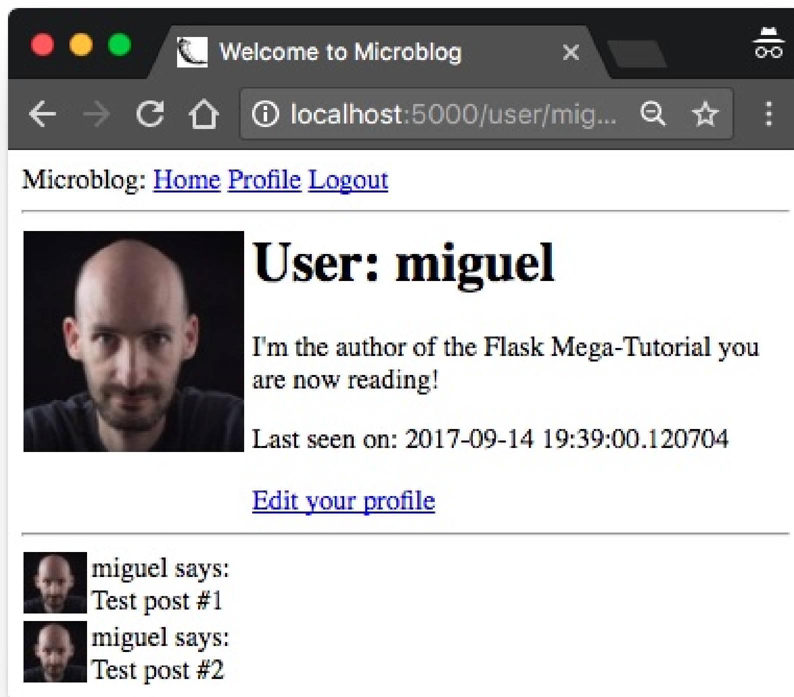
这个视图函数处理表单的方式和其他的视图函数略有不同。如果 `validate_on_submit()` 返回 `True` ，我将表单中的数据复制到用户对象中，然后将对象写入数据库。但是当 `validate_on_submit()` 返回 `False` 时，可能是由于两个不同的原因。这可能是因为浏览器刚刚发送了一个 `GET` 请求，我需要通过提供表单模板的初始版本来响应。也可能是这种情况，浏览器发送带有表单数据的 `POST` 请求，但该数据中的某些内容无效。对于该表单，我需要区别对待这两种情况。当第一次请求表单时，我用存储在数据库中的数据预填充字段，所以我需要做与提交相反的事情，那就是将存储在用户字段中的数据移动到表单中，这将确保这些表单字段具有用户的当前数据。但在验证错误的情况下，我不想写任何表单字段，因为它们已经由WTForms填充了。为了区分这两种情况，我需要检查 `request.method` ，如果它是 `GET` ，这是初始请求的情况，如果是 `POST` 则是提交表单验证失败的情况。



我将个人资料编辑页面的链接添加到个人主页，以便用户使用：

```
{% if user == current_user %}
<p><a href="{{ url_for('edit_profile') }}">Edit your profile</a></p>
{% endif %}
```

请注意我巧妙使用的条件，它确保在查看自己的个人主页时出现编辑个人资料的链接，而在查看其他人的个人主页时不会出现。


Welcome to Microblog
localhost:5000/user/mig...
Microblog: Home Profile Logout
User: miguel
I'm the author of the Flask Mega-Tutorial you are now reading!
Last seen on: 2017-09-14 19:39:00.120704
Edit your profile
miguel says:
Test post #1
miguel says:
Test post #2

本文翻译自The Flask Mega-Tutorial Part VII: Error Handling

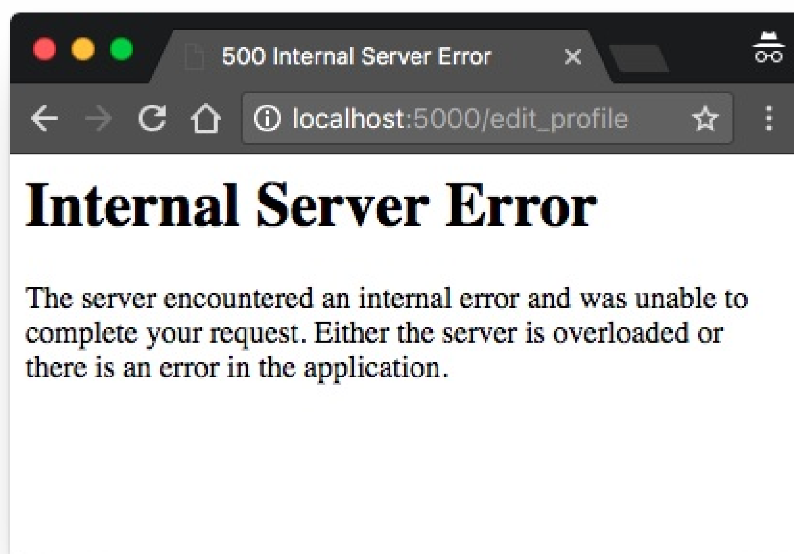
这是Flask Mega-Tutorial系列的第七部分，我将告诉你如何在Flask应用中进行错误处理。

本章将暂停为microblog应用开发新功能，转而讨论处理BUG的策略，因为它们总是无处不在。为了帮助本章的演示，我故意在第六章新增的代码中遗留了一处BUG。 在继续阅读之前，看看你能不能找到它！

本章的*GitHub*链接为：*Browse*, *Zip*, *Diff.*

Flask中的错误处理机制

在Flask应用中爆发错误时会发生什么？ 得到答案的最好的方法就是亲身体验一下。 启动应用，并确保至少有两个用户注册，以其中一个用户身份登录，打开个人主页并单击“编辑”链接。 在个人资料编辑器中，尝试将用户名更改为已经注册的另一个用户的用户名，boom！（爆炸声） 这将带来一个可怕的“Internal Server Error”页面：



如果你查看运行应用的终端会话，将看到stack trace（堆栈跟踪）。 堆栈跟踪在调试错误时非常有用，因为它们显示堆栈中调用的顺序，一直到产生错误的行：

```
(venv) $ flask run
 * Serving Flask app "microblog"
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
[2017-09-14 22:40:02,027] ERROR in app: Exception on /edit_profile [POST]
Traceback (most recent call last):
  File "/home/miguel/microblog/venv/lib/python3.6/site-packages/sqlalchemy/engine/base
.py", line 1182, in _execute_context
    context)
  File "/home/miguel/microblog/venv/lib/python3.6/site-packages/sqlalchemy/engine/defa
ult.py", line 470, in do_execute
    cursor.execute(statement, parameters)
sqlite3.IntegrityError: UNIQUE constraint failed: user.username
```

堆栈跟踪指示了BUG在何处。 本应用允许用户更改用户名，但却没有验证所选的新用户名与系统中已有的其他用户有没有冲突。 这个错误来自SQLAlchemy，它尝试将新的用户名写入数据库，但数据库拒绝了它，因为 `username` 列是用 `unique=True` 定义的。

值得注意的是，提供给用户的错误页面并没有提供关于错误的丰富信息，这是正确的做法。我绝对不希望用户知道崩溃是由数据库错误引起的，或者我正在使用什么数据库，或者是我的数据库中的一些表和字段名称。 所有这些信息都应该对外保密。

但是也有一些不尽人意之处。错误页面简陋不堪，与应用布局不匹配。 终端上的日志不断刷新，导致重要的堆栈跟踪信息被淹没，但我却需要不断回顾它，以免有漏网之鱼。 当然，我有一个BUG需要修复。 我将解决所有的这些问题，但首先，让我们来谈谈Flask的调试模式。

调试模式

你在上面看到的处理错误的方式对在生产服务器上运行的系统非常有用。 如果出现错误，用户将得到一个隐晦的错误页面（尽管我打算使这个错误页面更友好），错误的重要细节在服务器进程输出或存储到日志文件中。

但是当你正在开发应用时，可以启用调试模式，它是Flask在浏览器上直接运行一个友好调试器的模式。 要激活调试模式，请停止应用程序，然后设置以下环境变量：

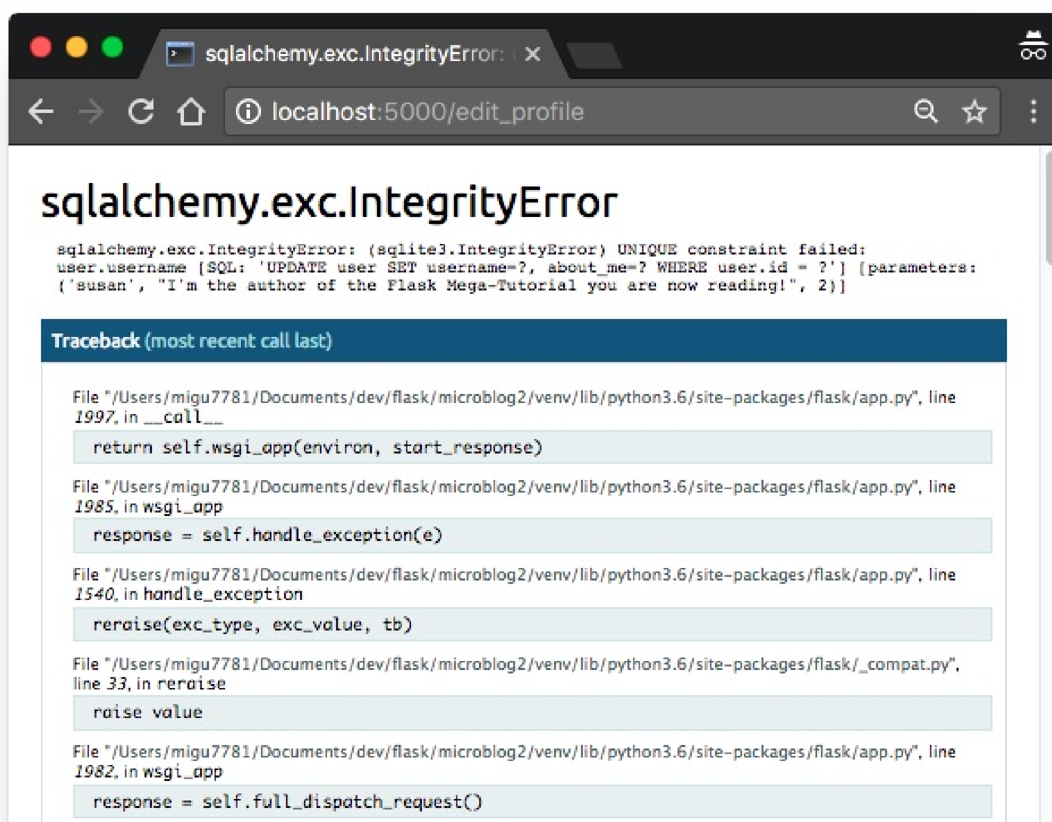
```
(venv) $ export FLASK_DEBUG=1
```

如果你使用Microsoft Windows，记得将 `export` 替换成 `set` 。

设置环境变量 `FLASK_DEBUG` 后，重启服务。相比之前，终端上的输出信息会有所变化：

```
(venv) microblog2 $ flask run
 * Serving Flask app "microblog"
 * Forcing debug mode on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 177-562-960
```

现在让应用再次崩溃，以在浏览器中查看交互式调试器：



该调试器允许你展开每个堆栈框来查看相应的源代码上下文。 你也可以在任意堆栈框上打开Python提示符并执行任何有效的Python表达式，例如检查变量的值。

永远不要在生产服务器上以调试模式运行Flask应用，这一点非常重要。 调试器允许用户远程执行服务器中的代码，因此对于想要渗入应用或服务器的恶意用户来说，这可能是开门揖盗。 作为附加的安全措施，运行在浏览器中的调试器开始被锁定，并且在第一次使用时会要求输入一个PIN码（你可以在 `flask run` 命令的输出中看到它）。

谈到调试模式的话题，我不得不提到的第二个重要的调试模式下的功能，就是重载器。 这是一个非常有用的开发功能，可以在源文件被修改时自动重启应用。 如果在调试模式下运行 `flask run` ，则可以在开发应用时，每当保存文件，应用都会重新启动以加载新的代码。

自定义错误页面

Flask为应用提供了一个机制来自定义错误页面，这样用户就不必看到简单而枯燥的默认页面。 作为例子，让我们为HTTP的404错误和500错误（两个最常见的错误页面）设置自定义错误页面。 为其他错误设置页面的方式与之相同。

使用 `@errorhandler` 装饰器来声明一个自定义的错误处理器。 我将把我的错误处理程序放在一个新的*app/errors.py*模块中。

```
from flask import render_template
from app import app, db

@app.errorhandler(404)
def not_found_error(error):
    return render_template('404.html'), 404

@app.errorhandler(500)
def internal_error(error):
    db.session.rollback()
    return render_template('500.html'), 500
```

错误函数与视图函数非常类似。 对于这两个错误，我将返回各自模板的内容。 请注意这两个函数在模板之后返回第二个值，这是错误代码编号。 对于之前我创建的所有视图函数，我不需要添加第二个返回值，因为我想要的是默认值200（成功响应的状态码）。 本处，这些是错误页面，所以我希望响应的状态码能够反映出来。

500错误的错误处理程序应当在引发数据库错误后调用，而上面的用户名重复实际上就是这种情况。 为了确保任何失败的数据库会话不会干扰模板触发的其他数据库访问，我执行会话回滚来将会话重置为干净的状态。

404错误的模板如下：

```
{% extends "base.html" %}

{% block content %}
    <h1>File Not Found</h1>
    <p><a href="{{ url_for('index') }}">Back</a></p>
{% endblock %}
```

500错误的模板如下：

```
{% extends "base.html" %}

{% block content %}
    <h1>An unexpected error has occurred</h1>
    <p>The administrator has been notified. Sorry for the inconvenience!</p>
    <p><a href="{{ url_for('index') }}">Back</a></p>
{% endblock %}
```

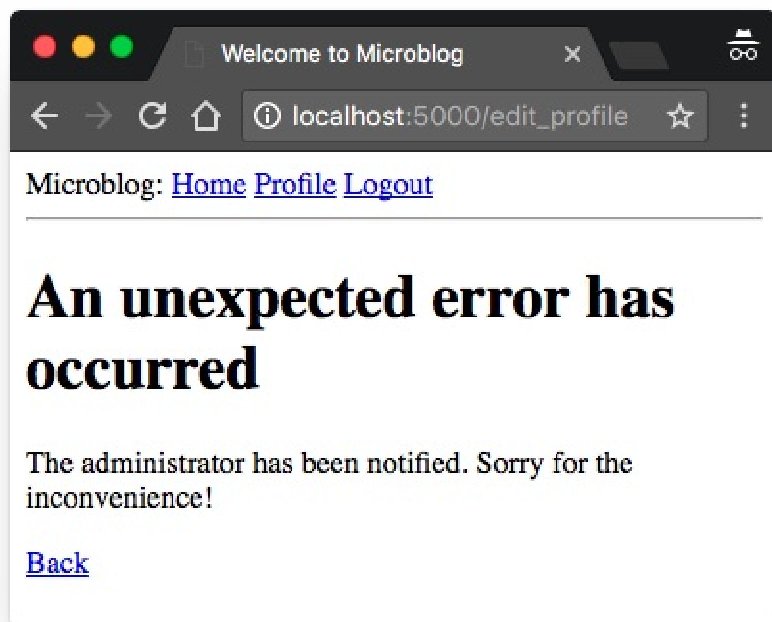
这两个模板都从 `base.html` 基础模板继承而来，所以错误页面与应用的普通页面有相同的外观布局。

为了让这些错误处理程序在Flask中注册，我需要在应用实例创建后导入新的*app/errors.py*模块：

```
# ...

from app import routes, models, errors
```

如果在终端界面设置环境变量 `FLASK_DEBUG=0` ，然后再次出发重复用户名的BUG，你将会看到一个更加友好的错误页面。



通过电子邮件发送错误

Flask提供的默认错误处理机制的另一个问题是没有通知机制，错误的堆栈跟踪只是被打印到终端，这意味着需要监视服务器进程的输出才能发现错误。 在开发时，这是非常好的，但是一旦将应用部署在生产服务器上，没有人会关心输出，因此需要采用更强大的解决方案。

我认为对错误发现采取积极主动的态度是非常重要的。 如果生产环境的应用发生错误，我想立刻知道。 所以我的第一个解决方案是配置Flask在发生错误之后立即向我发送一封电子邮件，邮件正文中包含错误堆栈跟踪的正文。

第一步，添加邮件服务器的信息到配置文件中：

```
class Config(object):
    # ...
    MAIL_SERVER = os.environ.get('MAIL_SERVER')
    MAIL_PORT = int(os.environ.get('MAIL_PORT') or 25)
    MAIL_USE_TLS = os.environ.get('MAIL_USE_TLS') is not None
    MAIL_USERNAME = os.environ.get('MAIL_USERNAME')
    MAIL_PASSWORD = os.environ.get('MAIL_PASSWORD')
    ADMINS = ['your-email@example.com']
```

电子邮件的配置变量包括服务器和端口，启用加密连接的布尔标记以及可选的用户名和密码。 这五个配置变量来源于环境变量。 如果电子邮件服务器没有在环境中设置，那么我将禁用电子邮件功能。 电子邮件服务器端口也可以在环境变量中给出，但是如果没有设置，则使用标准端口25。 电子邮件服务器凭证默认不使用，但可以根据需要提供。 `ADMINS` 配置变量是将收到错误报告的电子邮件地址列表，所以你自己的电子邮件地址应该在该列表中。

Flask使用Python的 `logging` 包来写它的日志，而且这个包已经能够通过电子邮件发送日志了。 我所需要做的就是为Flask的日志对象 `app.logger` 添加一个SMTPHandler的实例：

```
import logging
from logging.handlers import SMTPHandler

# ...

if not app.debug:
    if app.config['MAIL_SERVER']:
        auth = None
        if app.config['MAIL_USERNAME'] or app.config['MAIL_PASSWORD']:
            auth = (app.config['MAIL_USERNAME'], app.config['MAIL_PASSWORD'])
        secure = None
        if app.config['MAIL_USE_TLS']:
            secure = ()
        mail_handler = SMTPHandler(
            mailhost=(app.config['MAIL_SERVER'], app.config['MAIL_PORT']),
            fromaddr='no-reply@' + app.config['MAIL_SERVER'],
            toaddrs=app.config['ADMINS'], subject='Microblog Failure',
            credentials=auth, secure=secure)
        mail_handler.setLevel(logging.ERROR)
        app.logger.addHandler(mail_handler)
```

如你所见，仅当应用未以调试模式运行，且配置中存在邮件服务器时，我才会启用电子邮件日志记录器。

设置电子邮件日志记录器的步骤因为处理安全可选项而稍显繁琐。 本质上，上面的代码创建了一个 `SMTPHandler` 实例，设置它的级别，以便它只报告错误及更严重级别的信息，而不是警告，常规信息或调试消息，最后将它附加到Flask的 `app.logger` 对象中。

有两种方法来测试此功能。 最简单的就是使用Python的SMTP调试服务器。 这是一个模拟的电子邮件服务器，它接受电子邮件，然后打印到控制台。 要运行此服务器，请打开第二个终端会话并在其上运行以下命令：

```
(venv) $ python -m smtpd -n -c DebuggingServer localhost:8025
```

要用这个模拟邮件服务器来测试应用，那么你将设置 `MAIL_SERVER=localhost` 和 `MAIL_PORT=8025` 。

> 译者注：本段中去处了说明设置该端口需要管理员权限的部分，因为这和实际情况不符。原文如下： To test the application with this server, then you will set `MAIL_SERVER=localhost` and `MAIL_PORT=8025`. If you are on a Linux or Mac OS system, you will likely need to prefix the command with `sudo`, so that it can execute with administration privileges. If you are on a Windows system, you may need to open your terminal window as an administrator. Administrator rights are needed for this command because ports below 1024 are administrator-only ports. Alternatively, you can change the port to a higher port number, say 5025, and set `MAIL_PORT` variable to your chosen port in the environment, and that will not require administration rights.

保持调试SMTP服务器运行并返回到第一个终端，在环境中设置`export MAIL_SERVER=localhost`和`MAIL_PORT=8025`（如果使用的是Microsoft Windows，则使用set而不是export）。 确保FLASK_DEBUG变量设置为0或者根本不设置，因为应用不会在调试模式中发送电子邮件。 运行该应用并再次触发SQLAlchemy错误，以查看运行模拟电子邮件服务器的终端会话如何显示具有完整堆栈跟踪错误的电子邮件。

这个功能的第二个测试方法是配置一个真正的电子邮件服务器。 以下是使用你的Gmail帐户的电子邮件服务器的配置：

```
export MAIL_SERVER=smtp.googlemail.com
export MAIL_PORT=587
export MAIL_USE_TLS=1
export MAIL_USERNAME=<your-gmail-username>
export MAIL_PASSWORD=<your-gmail-password>
```

如果你使用的是Microsoft Windows，记住在每一条语句中用`set`替换掉`export`。

Gmail帐户中的安全功能可能会阻止应用通过它发送电子邮件，除非你明确允许“安全性较低的应用程序”访问你的Gmail帐户。 可以阅读此处来了解具体情况，如果你担心帐户的安全性，可以创建一个辅助邮箱帐户，配置它来仅用于测试电子邮件功能，或者你可以暂时启用允许不太安全的应用程序来运行此测试，完成后恢复为默认值。

记录日志到文件中

通过电子邮件来接收错误提示非常棒，但在其他场景下，有时候就有些不足了。有些错误条件既不是一个Python异常又不是重大事故，但是他们在调试的时候也是有足够用处的。为此，我将会为本应用维持一个日志文件。

为了启用另一个基于文件类型RotatingFileHandler的日志记录器，需要以和电子邮件日志记录器类似的方式将其附加到应用的logger对象中。

```
# ...
from logging.handlers import RotatingFileHandler
import os

# ...

if not app.debug:
    # ...

    if not os.path.exists('logs'):
        os.mkdir('logs')
    file_handler = RotatingFileHandler('logs/microblog.log', maxBytes=10240,
                                       backupCount=10)
    file_handler.setFormatter(logging.Formatter(
        '%(asctime)s %(levelname)s: %(message)s [in %(pathname)s:%(lineno)d]'))
    file_handler.setLevel(logging.INFO)
    app.logger.addHandler(file_handler)

    app.logger.setLevel(logging.INFO)
    app.logger.info('Microblog startup')
```

日志文件的存储路径位于顶级目录下，相对路径为 `logs/microblog.log` ，如果其不存在，则会创建它。

RotatingFileHandler类非常棒，因为它可以切割和清理日志文件，以确保日志文件在应用运行很长时间时不会变得太大。 本处，我将日志文件的大小限制为10KB，并只保留最后的十个日志文件作为备份。

logging.Formatter类为日志消息提供自定义格式。 由于这些消息正在写入到一个文件，我希望它们可以存储尽可能多的信息。 所以我使用的格式包括时间戳、日志记录级别、消息以及日志来源的源代码文件和行号。

为了使日志记录更有用，我还将应用和文件日志记录器的日志记录级别降低到 `INFO` 级别。 如果你不熟悉日志记录类别，则按照严重程度递增的顺序来认识它们就行了，分别是 `DEBUG` 、 `INFO` 、 `WARNING` 、 `ERROR` 和 `CRITICAL` 。

日志文件的第一个有趣用途是，服务器每次启动时都会在日志中写入一行。 当此应用在生产服务器上运行时，这些日志数据将告诉你服务器何时重新启动过。

修复用户名重复的BUG

利用用户名重复BUG这么久， 现在时候向你展示如何修复它了。

你是否还记得， `RegistrationForm` 已经实现了对用户名的验证，但是编辑表单的要求稍有不同。 在注册期间，我需要确保在表单中输入的用户名不存在于数据库中。 在编辑个人资料表单中，我必须做同样的检查，但有一个例外。 如果用户不改变原始用户名，那么验证应该允许，因为该用户名已经被分配给该用户 。 下面你可以看到我为这个表单实现了用户名验证：

```
class EditProfileForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired()])
    about_me = TextAreaField('About me', validators=[Length(min=0, max=140)])
    submit = SubmitField('Submit')

    def __init__(self, original_username, *args, **kwargs):
        super(EditProfileForm, self).__init__(*args, **kwargs)
        self.original_username = original_username

    def validate_username(self, username):
        if username.data != self.original_username:
            user = User.query.filter_by(username=self.username.data).first()
            if user is not None:
                raise ValidationError('Please use a different username.')
```

该实现使用了一个自定义的验证方法，接受表单中的用户名作为参数。 这个用户名保存为一个实例变量，并在`validate_username()`方法中被校验。 如果在表单中输入的用户名与原始用户名相同，那么就没有必要检查数据库是否有重复了。

为了使得新增的验证方法生效，我需要在对应视图函数中添加当前用户名到表单的username字段中：

```
@app.route('/edit_profile', methods=['GET', 'POST'])
@login_required
def edit_profile():
    form = EditProfileForm(current_user.username)
    # ...
```

现在这个BUG已经修复了，大多数情况下，以后在编辑个人资料时出现用户名重复的提交将被友好地阻止。 但这不是一个完美的解决方案，因为当两个或更多进程同时访问数据库时，这可能不起作用。假如存在验证通过的进程A和B都尝试修改用户名为同一个，但稍后进程A尝试重命名时，数据库已被进程B更改，无法重命名为该用户名，会再次引发数据库异常。除了有很多服务器进程并且非常繁忙的应用之外，这种情况是不太可能的，所以现在我不会为此担心。

此时，你可以尝试再次重现该错误，以了解新的表单验证方法如何防止该错误。

本文翻译自The Flask Mega-Tutorial Part VIII: Followers

这是Flask Mega-Tutorial系列的第八部分，我将告诉你如何实现类似于Twitter和其他社交网络的“粉丝”功能。

在本章中，我将更多地使用应用的数据库。 我希望应用的用户能够轻松便捷地关注其他用户。 所以我要扩展数据库，以便跟踪谁关注了谁，这比你想象的要难得多。

本章的*GitHub*链接为：*Browse*, *Zip*, *Diff*.

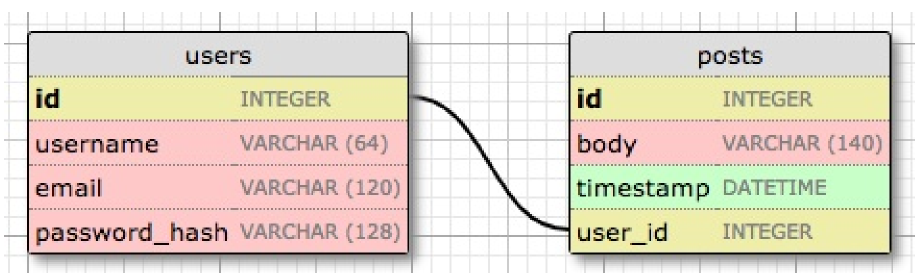
深入理解数据库关系

我上面说过，我想为每个用户维护一个“粉丝”用户列表和“关注”用户列表。 不幸的是，关系型数据库没有列表类型的字段来保存它们，那么只能通过表的现有字段和他们之间的关系来实现。

数据库已有一个代表用户的表，所以剩下的就是如何正确地组织他们之间的关注与被关注的关系。 这正是回顾基本数据库关系类型的好时机：

一对多

我已经在第四章中用过了一对多关系。这是该关系的示意图（译者注：实际表名分别为user和post）：



用户和用户动态通过这个关系来关联。其中，一个用户拥有多条用户动态，而一条用户动态属于一个用户（作者）。数据库在多的这方使用了一个外键以表示一对多关系。在上面的一对多关系中，外键是 `post` 表的 `user_id` 字段，这个字段将用户的每条动态都与其作者关联了起来。

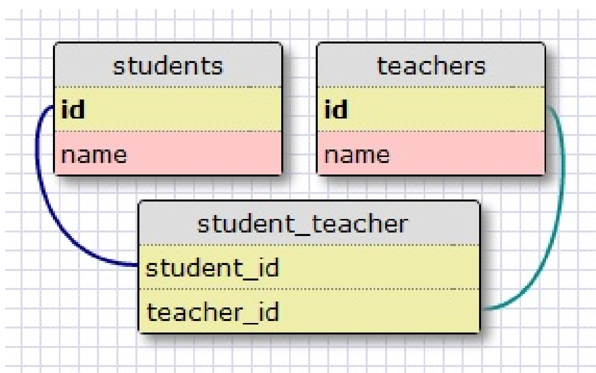
很明显， `user_id` 字段提供了直接访问给定用户动态的作者，但是反向呢？ 透过这层关系，我如何通过给定的用户来获得其用户动态的列表？ `post` 表中的 `user_id` 字段也足以回答这个问题，数据库具有索引，可以进行高效的查询“返回所有user_id字段等于X的用户动态”。

多对多

多对多关系会更加复杂，举个例子，数据库中有 `students` 表和 `teachers` 表，一名学生学习多位老师的课程，一位老师教授多名学生。这就像两个重叠的一对多关系。

对于这种类型的关系，我想要能够查询数据库来获取教授给定学生的教师的列表，以及某个教师课程中的学生的列表。 想要在关系型数据库中梳理这样的关系并非轻易而举，因为无法通过向现有表添加外键来完成此操作。

展现多对多关系需要使用额外的关联表。以下是数据库如何查找学生和教师的示例：



虽然起初看起来并不明显，但具有两个外键的关联表的确能够有效地回答所有多对多关系的查询。

多对一和一对一

多对一关系类似于一对多关系。 不同的是，这种关系是从“多”的角度来看的。

一对一的关系是一对多的特例。 实现是相似的，但是一个约束被添加到数据库，以防止“多”一方有多个链接。 虽然有这种类型的关系是有用的，但并不像其他类型那么普遍。

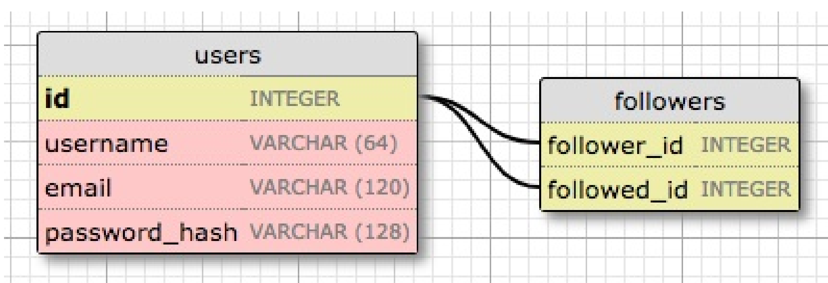
> 译者注：如果读者有兴趣，也可以看看我写的一篇类似的数据库关系文章——Web开发中常用的数据关系

实现粉丝机制

查看所有关系类型的概要，很容易确定维护粉丝关系的正确数据模型是多对多关系，因为用户可以关注多个其他用户，并且用户可以拥有多个粉丝。 不过，在学生和老师的例子中，多对多关系关联了两个实体。 但在粉丝关系中，用户关注其他用户，只有一个用户实体。 那么，多对多关系的第二个实体是什么呢？

该关系的第二个实体也是用户。 一个类的实例被关联到同一个类的其他实例的关系被称为自引用关系，这正是我在这里所用到的。

使用自引用多对多关系来实现粉丝机制的表结构示意图：



`followers` 表是关系的关联表。 此表中的外键都指向用户表中的数据行，因为它将用户关联到用户。 该表中的每个记录代表关注者和被关注者的一个关系。 像学生和老师的例子一样，像这样的设计允许数据库回答所有关于关注和被关注的问题，并且足够干净利落。

数据库模型的实现

首先，让我们在数据库中添加粉丝机制吧。这是 `followers` 关联表：

```
followers = db.Table('followers',
    db.Column('follower_id', db.Integer, db.ForeignKey('user.id')),
    db.Column('followed_id', db.Integer, db.ForeignKey('user.id'))
)
```

这是上图中关联表的直接翻译。 请注意，我没有像我为用户和用户动态所做的那样，将表声明为模型。 因为这是一个除了外键没有其他数据的辅助表，所以我创建它的时候没有关联到模型类。

现在我可以在用户表中声明多对多的关系了：

```
class User(UserMixin, db.Model):
    # ...
    followed = db.relationship(
        'User', secondary=followers,
        primaryjoin=(followers.c.follower_id == id),
        secondaryjoin=(followers.c.followed_id == id),
        backref=db.backref('followers', lazy='dynamic'), lazy='dynamic')
```

建立关系的过程实属不易。 就像我为 `post` 一对多关系所做的那样，我使用 `db.relationship` 函数来定义模型类中的关系。 这种关系将 `User` 实例关联到其他 `User` 实例，所以按照惯例，对于通过这种关系关联的一对用户来说，左侧用户关注右侧用户。 我在左侧的用户中定义了 `followed` 的关系，因为当我从左侧查询这个关系时，我将得到已关注的用户列表（即右侧的列表）。 让我们逐个检查这个 `db.relationship()` 所有的参数：

- `'User'` 是关系当中的右侧实体（将左侧实体看成是上级类）。由于这是自引用关系，所以我不得不在两侧都使用同一个实体。
- `secondary` 指定了用于该关系的关联表，就是使用我在上面定义的 `followers` 。
- `primaryjoin` 指明了通过关系表关联到左侧实体（关注者）的条件 。关系中的左侧的join条件是关系表中的 `follower_id` 字段与这个关注者的用户ID匹

配。`followers.c.follower_id`表达式引用了该关系表中的`follower_id`列。
- `secondaryjoin` 指明了通过关系表关联到右侧实体（被关注者）的条件 。 这个条件与`primaryjoin`类似，唯一的区别在于，现在我使用关系表的字段的是`followed_id`了。
- `backref`定义了右侧实体如何访问该关系。在左侧，关系被命名为`followed`，所以在右侧我将使用`followers`来表示所有左侧用户的列表，即粉丝列表。附加的`lazy`参数表示这个查询的执行模式，设置为动态模式的查询不会立即执行，直到被调用，这也是我设置用户动态一对多的关系的方式。
- `lazy`和`backref`中的`lazy`类似，只不过当前的这个是应用于左侧实体，`backref`中的是应用于右侧实体。

如果理解起来比较困难，你也不必过于担心。我待会儿就会向你展示如何利用这些关系来执行查询，一切就会变得清晰明了。

数据库的变更，需要记录到一个新的数据库迁移中：

```
(venv) $ flask db migrate -m "followers"
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.autogenerate.compare] Detected added table 'followers'
  Generating /home/miguel/microblog/migrations/versions/ae346256b650_followers.py ...
done

(venv) $ flask db upgrade
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade 37f06a334dbf -> ae346256b650, follow
ers
```

关注和取消关注

感谢SQLAlchemy ORM，一个用户关注另一个用户的行为可以通过`followed`关系抽象成一个列表来简便使用。 例如，如果我有两个用户存储在`user1`和`user2`变量中，我可以用下面这个简单的语句来实现：

```
user1.followed.append(user2)
```

要取消关注该用户，我可以这么做：

```
user1.followed.remove(user2)
```

即便关注和取消关注的操作相当容易，我仍然想提高这段代码的可重用性，所以我不会直接在代码中使用“appends”和“removes”，取而代之，我将在`User`模型中实现“follow”和“unfollow”方法。 最好将应用逻辑从视图函数转移到模型或其他辅助类或辅助模块中，因为你会在本章之后将会看到，这使得单元测试更加容易。

下面是用户模型中添加和删除关注关系的代码变更：

```
class User(UserMixin, db.Model):
    #...

    def follow(self, user):
        if not self.is_following(user):
            self.followed.append(user)

    def unfollow(self, user):
        if self.is_following(user):
            self.followed.remove(user)

    def is_following(self, user):
        return self.followed.filter(
            followers.c.followed_id == user.id).count() > 0
```

`follow()` 和 `unfollow()` 方法使用关系对象的 `append()` 和 `remove()` 方法。有必要在处理关系之前，使用一个 `is_following()` 方法来确认操作的前提条件是否符合，例如，如果我要求 `user1` 关注 `user2` ，但事实证明这个关系在数据库中已经存在，我就没必要重复操作了。相同的逻辑可以应用于取消关注。

`is_following()` 方法发出一个关于 `followed` 关系的查询来检查两个用户之间的关系是否已经存在。 你已经看到过我使用SQLAlchemy查询对象的 `filter_by()` 方法，例如，查找给定用户名的用户。 我在这里使用的 `filter()` 方法很类似，但是更加偏向底层，因为它可以包含任意的过滤条件，而不像 `filter_by()` ，它只能检查是否等于一个常量值。 我在 `is_following()` 中使用的过滤条件是，查找关联表中左侧外键设置为 `self` 用户且右侧设置为 `user` 参数的数据行。 查询以 `count()` 方法结束，返回结果的数量。 这个查询的结果是 `0` 或 `1` ，因此检查计数是1还是大于0实际上是相等的。 至于其他的查询结束符 `all()` 和 `first()` ，你已经看到我使用过了。

查看已关注用户的动态

在数据库中支持粉丝机制的工作几近尾声，但是我却遗漏了一项重要的功能。应用主页中需要展示已登录用户关注的其他所有用户的动态，我需要用数据库查询来返回这些用户动态。

最显而易见的方案是先执行一个查询以返回已关注用户的列表，如你所知，可以使用 `user.followed.all()` 语句。然后对每个已关注的用户执行一个查询来返回他们的用户动态。最后将所有用户的动态按照日期时间倒序合并到一个列表中。听起来不错？其实不然。

这种方法有几个问题。 如果一个用户关注了一千人，会发生什么？ 我需要执行一千个数据库查询来收集所有的用户动态。 然后我需要合并和排序内存中的一千个列表。 作为第二个问题，考虑到应用主页最终将实现分页，所以它不会显示所有可用的用户动态，只能是前几个，并显示一个链接来提供感兴趣的用户查看更多动态。 如果我要按它们的日期排序来显示动态，我怎么能知道哪些用户动态才是所有用户中最新的呢？除非我首先得到了所有的用户动态并对其进行排序。 这实际上是一个糟糕的解决方案，不能很好地应对规模化。

用户动态的合并和排序操作是无法避免的，但是在应用中执行会导致效率十分低下， 而这种工作是关系数据库擅长的。 我可以使用数据库的索引，命令它以更有效的方式执行查询和排序。 所以我真正想要提供的方案是，定义我想要得到的信息来执行一个数据库查询，然后让数据库找出如何以最有效的方式来提取这些信息。

看看下面的这个查询：

```
class User(db.Model):
    #...
    def followed_posts(self):
        return Post.query.join(
            followers, (followers.c.followed_id == Post.user_id)).filter(
                followers.c.follower_id == self.id).order_by(
                    Post.timestamp.desc())
```

这是迄今为止我在这个应用中使用的最复杂的查询。 我将尝试一步一步地解读这个查询。 如果你看一下这个查询的结构，你会注意到有三个主要部分，分别是`join()`、`filter()`和`order_by()`，他们都是SQLAlchemy查询对象的方法：

```
Post.query.join(...).filter(...).order_by(...)
```

联合查询

要理解join操作的功能，我们来看一个例子。 假设我有一个包含以下内容的`User`表：

id	username
1	john
2	susan
3	mary
4	david

为了简单起见，我只会保留用户模型的`id`和`username`字段以便进行查询，其他的都略去。

假设`followers`关系表中数据表达的是用户`john`关注用户`susan`和`david`，用户`susan`关注用户`mary`，用户`mary`关注用户`david`。这些的数据如下表所示：

follower_id	followed_id
1	2
1	4
2	3
3	4

最后，用户动态表中包含了每个用户的一条动态：

id	text	user_id
1	post from susan	2
2	post from mary	3
3	post from david	4
4	post from john	1

这张表也省略了一些不属于这个讨论范围的字段。

这是我为该查询再次设计的`join()`调用：

```
Post.query.join(followers, (followers.c.followed_id == Post.user_id))
```

我在用户动态表上调用join操作。 第一个参数是followers关联表，第二个参数是join条件。 我的这个调用表达的含义是我希望数据库创建一个临时表，它将用户动态表和关注者表中的数据结合在一起。 数据将根据参数传递的条件进行合并。

我使用的条件表示了followers关系表的`followed_id`字段必须等于用户动态表的`user_id`字段。 要执行此合并，数据库将从用户动态表（join的左侧）获取每条记录，并追加`followers`关系表（join的右侧）中的匹配条件的所有记录。 如果`followers`关系表中有多个记录符合条件，那么用户动态数据行将重复出现。 如果对于一个给定的用户动态，followers关系表中却没有匹配，那么该用户动态的记录不会出现在join操作的结果中。

利用我上面定义的示例数据，执行join操作的结果如下：

id	text	user_id	follower_id	followed_id
1	post from susan	2	1	2
2	post from mary	3	2	3
3	post from david	4	1	4
3	post from david	4	3	4

注意`user_id`和`followed_id`列在所有数据行中都是相等的，因为这是join条件。 来自用户`john`的用户动态不会出现在临时表中，因为被关注列表中没有包含`john`用户，换句话说，没有任何人关注john。 而来自`david`的用户动态出现了两次，因为该用户有两个粉丝。

虽然创建了这个join操作，但却没有得到想要的结果。请继续看下去，因为这只是更大的查询的一部分。

过滤

Join操作给了我一个所有被关注用户的用户动态的列表，远超出我想要的那部分数据。 我只对这个列表的一个子集感兴趣——某个用户关注的用户们的动态，所以我需要用 `filter()` 来剔除所有我不需要的数据。

这是过滤部分的查询语句：

```
filter(followers.c.follower_id == self.id)
```

该查询是 `User` 类的一个方法， `self.id` 表达式是指我感兴趣的用户的ID。 `filter()` 挑选临时表中 `follower_id` 列等于这个ID的行，换句话说，我只保留follower(粉丝)是该用户的数据。

假如我现在对 `id` 为 1 的用户 `john` 能看到的用户动态感兴趣，这是从临时表过滤后的结果：

id	text	user_id	follower_id	followed_id
1	post from susan	2	1	2
3	post from david	4	1	4

这正是我想要的结果！

请记住，查询是从 `Post` 类中发出的，所以尽管我曾经得到了由数据库创建的一个临时表来作为查询的一部分，但结果将是包含在此临时表中的用户动态， 而不会存在由于执行join操作添加的其他列。

排序

查询流程的最后一步是对结果进行排序。这部分的查询语句如下：

```
order_by(Post.timestamp.desc())
```

在这里，我要说的是，我希望使用用户动态产生的时间戳按降序排列结果列表。排序之后，第一个结果将是最新的用户动态。

组合自身动态和关注的用户动态

我在 `followed_posts()` 函数中使用的查询是非常有用的，但有一个限制，人们期望看到他们自己的动态包含在他们的关注的用户动态的时间线中，而该查询却力有未逮。

有两种可能的方式来扩展此查询以包含用户自己的动态。 最直截了当的方法是将查询保持原样，但要确保所有用户都关注了他们自己。 如果你是你自己的粉丝，那么上面的查询就会找到你自己的动态以及你关注的所有人的动态。 这种方法的缺点是会影响粉丝的统计数据。 所

有人的粉丝数量都将加一，所以它们必须在显示之前进行调整。 第二种方法是通过创建第二个查询返回用户自己的动态，然后使用“union”操作将两个查询合并为一个查询。

深思熟虑之后，我选择了第二个方案。 下面你可以看到 `followed_posts()` 函数已被扩展成通过联合查询来并入用户自己的动态：

```
    def followed_posts(self):
        followed = Post.query.join(
            followers, (followers.c.followed_id == Post.user_id)).filter(
                followers.c.follower_id == self.id)
        own = Post.query.filter_by(user_id=self.id)
        return followed.union(own).order_by(Post.timestamp.desc())
```

请注意， `followed` 和 `own` 查询结果集是在排序之前进行的合并。

对用户模型执行单元测试

虽然我不担心这个稍显“复杂”的粉丝机制的运行是否无误。 但当我编写举足轻重的代码时，我担心的是我在应用的不同部分修改了代码之后，如何确保本处代码将来会继续工作。 确保已经编写的代码在将来继续有效的最佳方法是创建一套自动化测试，你可以在每次更新代码后执行测试。

Python包含一个非常有用的 `unittest` 包，可以轻松编写和执行单元测试。 让我们来为 `User` 类中的现有方法编写一些单元测试并存储到*tests.py*模块：

```
from datetime import datetime, timedelta
import unittest
from app import app, db
from app.models import User, Post

class UserModelCase(unittest.TestCase):
    def setUp(self):
        app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite://'
        db.create_all()

    def tearDown(self):
        db.session.remove()
        db.drop_all()

    def test_password_hashing(self):
        u = User(username='susan')
        u.set_password('cat')
        self.assertFalse(u.check_password('dog'))
        self.assertTrue(u.check_password('cat'))

    def test_avatar(self):
        u = User(username='john', email='john@example.com')
        self.assertEqual(u.avatar(128), ('https://www.gravatar.com/avatar/'
                                         'd4c74594d841139328695756648b6bd6'
                                         '?d=identicon&s=128'))

    def test_follow(self):
        u1 = User(username='john', email='john@example.com')
        u2 = User(username='susan', email='susan@example.com')
        db.session.add(u1)
        db.session.add(u2)
        db.session.commit()
```

```
        self.assertEqual(u1.followed.all(), [])
        self.assertEqual(u1.followers.all(), [])

        u1.follow(u2)
        db.session.commit()
        self.assertTrue(u1.is_following(u2))
        self.assertEqual(u1.followed.count(), 1)
        self.assertEqual(u1.followed.first().username, 'susan')
        self.assertEqual(u2.followers.count(), 1)
        self.assertEqual(u2.followers.first().username, 'john')

        u1.unfollow(u2)
        db.session.commit()
        self.assertFalse(u1.is_following(u2))
        self.assertEqual(u1.followed.count(), 0)
        self.assertEqual(u2.followers.count(), 0)

    def test_follow_posts(self):
        # create four users
        u1 = User(username='john', email='john@example.com')
        u2 = User(username='susan', email='susan@example.com')
        u3 = User(username='mary', email='mary@example.com')
        u4 = User(username='david', email='david@example.com')
        db.session.add_all([u1, u2, u3, u4])

        # create four posts
        now = datetime.utcnow()
        p1 = Post(body="post from john", author=u1,
                  timestamp=now + timedelta(seconds=1))
        p2 = Post(body="post from susan", author=u2,
                  timestamp=now + timedelta(seconds=4))
        p3 = Post(body="post from mary", author=u3,
                  timestamp=now + timedelta(seconds=3))
        p4 = Post(body="post from david", author=u4,
                  timestamp=now + timedelta(seconds=2))
        db.session.add_all([p1, p2, p3, p4])
        db.session.commit()

        # setup the followers
        u1.follow(u2)  # john follows susan
        u1.follow(u4)  # john follows david
        u2.follow(u3)  # susan follows mary
        u3.follow(u4)  # mary follows david
        db.session.commit()

        # check the followed posts of each user
        f1 = u1.followed_posts().all()
        f2 = u2.followed_posts().all()
        f3 = u3.followed_posts().all()
        f4 = u4.followed_posts().all()
        self.assertEqual(f1, [p2, p4, p1])
        self.assertEqual(f2, [p2, p3])
        self.assertEqual(f3, [p3, p4])
        self.assertEqual(f4, [p4])

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

我添加了四个用户模型的测试，包含密码哈希、用户头像和粉丝功能。

`setUp()` 和 `tearDown()` 方法是单元测试框架分别在每个测试之前和之后执行的特殊方法。 我在 `setUp()` 中实现了一些小技巧，以防止单元测试使用我用于开发的常规数据库。 通过将应用配置更改为 `sqlite://` ，我在测试过程中通过SQLAlchemy来使用SQLite内存数据库。 `db.create_all()` 创建所有的数据库表。 这是从头开始创建数据库的快速方法，在测试中相当好用。 而对于开发环境和生产环境的数据库结构管理，我已经通过数据库迁移的手段向你展示过了。

你可以使用以下命令运行整个测试组件：

```
(venv) $ python tests.py
test_avatar (__main__.UserModelCase) ... ok
test_follow (__main__.UserModelCase) ... ok
test_follow_posts (__main__.UserModelCase) ... ok
test_password_hashing (__main__.UserModelCase) ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.494s

OK
```

从现在起，每次对应用进行更改时，都可以重新运行测试，以确保正在测试的功能没有受到影响。 另外，每次将另一个功能添加到应用时，都应该为其编写一个单元测试。

在应用中集成粉丝机制

数据库和模型中粉丝机制的实现现在已经完成，但是我没有将它集成到应用中，所以我现在要添加这个功能。 值得高兴的是，实现它没有什么大的挑战，都将基于你已经学过的概念。

让我们来添加两个新的路由和视图函数，它们提供了用户关注和取消关注的URL和逻辑实现：

```
@app.route('/follow/<username>')
@login_required
def follow(username):
    user = User.query.filter_by(username=username).first()
    if user is None:
        flash('User {} not found.'.format(username))
        return redirect(url_for('index'))
    if user == current_user:
        flash('You cannot follow yourself!')
        return redirect(url_for('user', username=username))
    current_user.follow(user)
    db.session.commit()
    flash('You are following {}!'.format(username))
    return redirect(url_for('user', username=username))

@app.route('/unfollow/<username>')
@login_required
def unfollow(username):
    user = User.query.filter_by(username=username).first()
    if user is None:
        flash('User {} not found.'.format(username))
        return redirect(url_for('index'))
    if user == current_user:
        flash('You cannot unfollow yourself!')
        return redirect(url_for('user', username=username))
    current_user.unfollow(user)
    db.session.commit()
    flash('You are not following {}.'.format(username))
    return redirect(url_for('user', username=username))
```

视图函数的逻辑不言而喻，但要注意所有的错误检查，以防止出现意外的问题，并尝试在出现问题时向用户提供有用的信息。

我将添加这两个视图函数的路由到每个用户的个人主页中，以便其他用户执行关注和取消关注的操作：

```
        ...
        <h1>User: {{ user.username }}</h1>
        {% if user.about_me %}<p>{{ user.about_me }}</p>{% endif %}
        {% if user.last_seen %}<p>Last seen on: {{ user.last_seen }}</p>{% endif %}
        <p>{{ user.followers.count() }} followers, {{ user.followed.count() }} followi
ng.</p>
        {% if user == current_user %}
        <p><a href="{{ url_for('edit_profile') }}">Edit your profile</a></p>
        {% elif not current_user.is_following(user) %}
        <p><a href="{{ url_for('follow', username=user.username) }}">Follow</a></p>
        {% else %}
        <p><a href="{{ url_for('unfollow', username=user.username) }}">Unfollow</a></p
>
        {% endif %}
        ...
```

用户个人主页的变更，一是在最近访问的时间戳之下添加一行，以显示此用户拥有多少个粉丝和关注用户。二是当你查看自己的个人主页时出现的“Edit”链接的行，可能会变成以下三种链接之一：

- 如果用户查看他(她)自己的个人主页，仍然是“Edit”链接不变。
- 如果用户查看其他并未关注的用户的个人主页，显示“Follow”链接。
- 如果用户查看其他已经关注的用户的个人主页，显示“Unfollow”链接。

此时，你可以运行该应用，创建一些用户并测试一下关注和取消关注用户的功能。 唯一需要记住的是，需要手动键入你要关注或取消关注的用户的个人主页URL，因为目前没有办法查看用户列表。 例如，如果你想关注`susan`，则需要在浏览器的地址栏中输入*http://localhost:5000/user/susan*以访问该用户的个人主页。 请确保你在测试关注和取消关注的时候，留意到了其粉丝和关注的数量变化。

我应该在应用的主页上显示用户动态的列表，但是我还没有完成所有依赖的工作，因为用户不能发表动态。 所以我会暂缓这个页面的完善工作，直到发表用户动态功能的完成。

本文翻译自The Flask Mega-Tutorial Part IX: Pagination

这是Flask Mega-Tutorial系列的第九部分，我将告诉你如何对数据列表进行分页。

在第八章我已经做了几个数据库更改，以支持在社交网络非常流行的“粉丝”机制。 有了这个功能，接下来我准备好删除一开始就使用的模拟用户动态了。 在本章中，应用将开始接受来自用户的动态更新，并将其发布到网站首页和个人主页。

本章的*GitHub*链接为：*Browse*, *Zip*, *Diff.*

发布用户动态

让我们从简单的事情开始吧。 首页需要有一个表单，用户可以在其中键入新动态。 我创建一个表单类：

```
class PostForm(FlaskForm):
    post = TextAreaField('Say something', validators=[
        DataRequired(), Length(min=1, max=140)])
    submit = SubmitField('Submit')
```

然后，我将该表单添加到网站首页的模板中：

```
{% extends "base.html" %}

{% block content %}
    <h1>Hi, {{ current_user.username }}!</h1>
    <form action="" method="post">
        {{ form.hidden_tag() }}
        <p>
            {{ form.post.label }}<br>
            {{ form.post(cols=32, rows=4) }}<br>
            {% for error in form.post.errors %}
            <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <p>{{ form.submit() }}</p>
    </form>
    {% for post in posts %}
    <p>
    {{ post.author.username }} says: <b>{{ post.body }}</b>
    </p>
    {% endfor %}
{% endblock %}
```

模板中的变更和处理以前的表单类似。最后的部分是将表单处理逻辑添加到视图函数中：

```
from app.forms import PostForm
from app.models import Post

@app.route('/', methods=['GET', 'POST'])
@app.route('/index', methods=['GET', 'POST'])
@login_required
def index():
    form = PostForm()
    if form.validate_on_submit():
        post = Post(body=form.post.data, author=current_user)
        db.session.add(post)
        db.session.commit()
        flash('Your post is now live!')
        return redirect(url_for('index'))
    posts = [
        {
            'author': {'username': 'John'},
            'body': 'Beautiful day in Portland!'
        },
        {
            'author': {'username': 'Susan'},
            'body': 'The Avengers movie was so cool!'
        }
    ]
    return render_template("index.html", title='Home Page', form=form,
                           posts=posts)
```

我们来一个个地解读该视图函数的变更：

- 导入`Post`和`PostForm`类
- 关联到`index`视图函数的两个路由都新增接受`POST`请求，以便视图函数处理接收的表单数据
- 处理表单的逻辑会为`post`表插入一条新的数据
- 模板新增接受`form`对象，以便渲染文本输入框

在继续之前，我想提一些与Web表单处理相关的重要内容。 请注意，在处理表单数据后，我通过发送重定向到主页来结束请求。 我可以轻松地跳过重定向，并允许函数继续向下进入模板渲染部分，因为这已经是主页视图函数了。

那么，为什么重定向呢？ 通过重定向来响应Web表单提交产生的POST请求是一种标准做法。 这有助于缓解在Web浏览器中执行刷新命令的烦恼。 当你点击刷新键时，所有的网页浏览器都会重新发出最后的请求。 如果带有表单提交的POST请求返回一个常规的响应，那么刷新将重新提交表单。 因为这不是预期的行为，所以浏览器会要求用户确认重复的提交，但是大多数用户却很难理解浏览器询问的内容。不过，如果一个`POST`请求被重定向响应，浏览器现在被指示发送`GET`请求来获取重定向中指定的页面，所以现在最后一个请求不再是'POST'请求了， 刷新命令就能以更可预测的方式工作。

这个简单的技巧叫做Post/Redirect/Get模式。 它避免了用户在提交网页表单后无意中刷新页面时插入重复的动态。

展示用户动态

如果你还记得，我创建过几条模拟的用户动态，展示在主页已经有一段时间了。 这些模拟对象是在 `index` 视图函数中显式创建的一个简单的Python列表：

```
    posts = [
        {
            'author': {'username': 'John'},
            'body': 'Beautiful day in Portland!'
        },
        {
            'author': {'username': 'Susan'},
            'body': 'The Avengers movie was so cool!'
        }
    ]
```

但是现在我在 `User` 模型中有了 `followed_posts()` 方法，它可以返回给定用户希望看到的用户动态的查询结果集。 所以现在我可以用真正的用户动态替换模拟的用户动态：

```
@app.route('/', methods=['GET', 'POST'])
@app.route('/index', methods=['GET', 'POST'])
@login_required
def index():
    # ...
    posts = current_user.followed_posts().all()
    return render_template("index.html", title='Home Page', form=form,
                           posts=posts)
```

`User` 类的 `followed_posts` 方法返回一个SQLAlchemy查询对象，该对象被配置为从数据库中获取用户感兴趣的用户动态。 在这个查询中调用 `all()` 会触发它的执行，返回值是包含所有结果的列表。 所以我最终得到了一个与我迄今为止一直使用的模拟用户动态非常相似的结构。 它们非常接近，模板甚至不需要改变。

更容易地发现和关注用户

相信你已经留意到了，应用没有一个很好的途径来让用户可以找到其他用户进行关注。实际上，现在根本没有办法在页面上查看到底有哪些用户存在。我将会使用少量简单的变更来解决这个问题。

我将会创建一个新的“发现”页面。该页面看起来像是主页，但是却不是只显示已关注用户的动态，而是展示所有用户的全部动态。新增的发现视图函数如下：

```
@app.route('/explore')
@login_required
def explore():
    posts = Post.query.order_by(Post.timestamp.desc()).all()
    return render_template('index.html', title='Explore', posts=posts)
```

你有没有注意到这个视图函数中的奇怪之处？ `render_template()` 引用了我在应用的主页面中使用的*index.html*模板。 这个页面与主页非常相似，所以我决定重用这个模板。 但与主页不同的是，在发现页面不需要一个发表用户动态表单，所以在这个视图函数中，我没有在模板

调用中包含 `form` 参数。

要防止*index.html*模板在尝试呈现不存在的Web表单时崩溃，我将添加一个条件，只在传入表单参数后才会呈现该表单：

```
{% extends "base.html" %}

{% block content %}
    <h1>Hi, {{ current_user.username }}!</h1>
    {% if form %}
    <form action="" method="post">
        ...
    </form>
    {% endif %}
    ...
{% endblock %}
```

该页面也需要添加到导航栏中：

```
        <a href="{{ url_for('explore') }}">Explore</a>
```

还记得我在第六章中介绍的用于个人主页渲染用户动态的*_post.html*子模板吗？ 这是一个包含在个人主页模板中的小模板，它独立于其他模板，因此也可以被这些模板调用。 我现在要做一个小小的改进，将用户动态作者的用户名显示为一个链接：

```
    <table>
        <tr valign="top">
            <td><img src="{{ post.author.avatar(36) }}"></td>
            <td>
                <a href="{{ url_for('user', username=post.author.username) }}">
                    {{ post.author.username }}
                </a>
                says:<br>{{ post.body }}
            </td>
        </tr>
    </table>
```

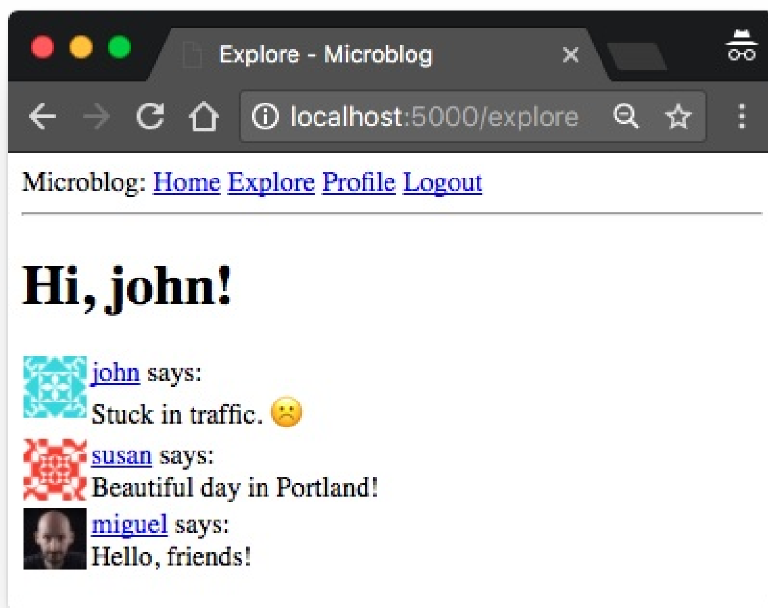
然后在主页和发现页中使用这个子模板来渲染用户动态：

```
    ...
    {% for post in posts %}
        {% include '_post.html' %}
    {% endfor %}
    ...
```

子模板期望存在一个名为 `post` 的变量，才能正常工作。该变量是上层模板中通过循环产生的。

通过这些细小的变更，应用的用户体验得到了大大的提升。现在，用户可以访问发现页来查看陌生用户的动态，并通过这些用户动态来关注用户，而需要的操作仅仅是点击用户名跳转到其个人主页并点击关注链接。令人叹为观止！对吧？

此时，我建议你在应用上再次尝试一下这个功能，以便体验最后的用户接口的完善。



用户动态的分页

应用看起来更完善了，但是在主页显示所有用户动态迟早会出问题。如果一个用户有成千上万条关注的用户动态时，会发生什么？你可以想象得到，管理这么大的用户动态列表将会变得相当缓慢和低效。

为了解决这个问题，我会将用户动态进行分页。这意味着一开始显示的只是所有用户动态的一部分，并提供链接来访问其余的用户动态。Flask-SQLAlchemy的 `paginate()` 方法原生就支持分页。例如，我想要获取用户关注的前20个动态，我可以将 `all()` 结束调用替换成如下的查询：

```
>>> user.followed_posts().paginate(1, 20, False).items
```

Flask-SQLAlchemy的所有查询对象都支持 `paginate` 方法，需要输入三个参数来调用它：

- 从1开始的页码
- 每页的数据量
- 错误处理布尔标记，如果是 `True` ，当请求范围超出已知范围时自动引发404错误。如果是 `False` ，则会返回一个空列表。

`paginate` 方法返回一个 `Pagination` 的实例。其 `items` 属性是请求内容的数据列表。 `Pagination` 实例还有一些其他用途，我会在之后讨论。

现在想想如何在`index()`视图函数展现分页呢。我先来给应用添加一个配置项，以表示每页展示的数据列表长度吧。

```
class Config(object):
    # ...
    POSTS_PER_PAGE = 3
```

存储这些应用范围的“可控机关”到配置文件是一个好主意，因为这样我调整时只需去一个地方。在最终的应用中，每页显示的数据将会大于三，但是对于测试而言，使用小数字很方便。

接下来，我需要决定如何将页码并入到应用URL中。一个相当常见的方法是使用查询字符串参数来指定一个可选的页码，如果没有给出则默认为页面1。以下是一些示例网址，显示了我将如何实现这一点：

- 第1页，隐含：*http://localhost:5000/index*
- 第1页，显式：*http://localhost:5000/index?page=1*
- 第3页：*http://localhost:5000/index?page=3*

要访问查询字符串中给出的参数，我可以使用Flask的request.args对象。你已经在第五章中看到了这种方法，我用Flask-Login实现了用户登录的可以包含一个`next`查询字符串参数的URL。

给主页和发现页的视图函数添加分页的代码变更如下：

```
@app.route('/', methods=['GET', 'POST'])
@app.route('/index', methods=['GET', 'POST'])
@login_required
def index():
    # ...
    page = request.args.get('page', 1, type=int)
    posts = current_user.followed_posts().paginate(
        page, app.config['POSTS_PER_PAGE'], False)
    return render_template('index.html', title='Home', form=form,
                           posts=posts.items)

@app.route('/explore')
@login_required
def explore():
    page = request.args.get('page', 1, type=int)
    posts = Post.query.order_by(Post.timestamp.desc()).paginate(
        page, app.config['POSTS_PER_PAGE'], False)
    return render_template("index.html", title='Explore', posts=posts.items)
```

通过这些更改，这两个路由决定了要显示的页码，可以从`page`查询字符串参数获得或是默认值1。然后使用`paginate()`方法来检索指定范围的结果。决定页面数据列表大小的`POSTS_PER_PAGE`配置项是通过`app.config`对象中获取的。

请注意，这些更改非常简单，每次更改都只会影响很少的代码。我试图在编写应用每个部分的时候，不做任何有关其他部分如何工作的假设，这使我可以编写更易于扩展和测试的且兼具模块化和健壮性的应用，并且不太可能失败或出现BUG。

来尝试下分页功能吧。首先确保你有三条以上的用户动态。在发现页面中更方便测试，因为该页面显示所有用户的动态。你现在只会看到最近的三条用户动态。如果你想看接下来的三条，请在浏览器的地址栏中输入*http://localhost:5000/explore?page=2*。

分页导航

接下来的改变是在用户动态列表的底部添加链接，允许用户导航到下一页或上一页。还记得我曾提到过 `paginate()` 的返回是 `Pagination` 类的实例吗？到目前为止，我已经使用了此对象的 `items` 属性，其中包含为所选页面检索的用户动态列表。但是这个分页对象还有一些其他的属性在构建分页链接时很有用：

- `has_next`：当前页之后存在后续页面时为真
- `has_prev`：当前页之前存在前置页面时为真
- `next_num`：下一页的页码
- `prev_num`：上一页的页码

有了这四个元素，我就可以生成上一页和下一页的链接并将其传入模板以渲染：

```
@app.route('/', methods=['GET', 'POST'])
@app.route('/index', methods=['GET', 'POST'])
@login_required
def index():
    # ...
    page = request.args.get('page', 1, type=int)
    posts = current_user.followed_posts().paginate(
        page, app.config['POSTS_PER_PAGE'], False)
    next_url = url_for('index', page=posts.next_num) \
        if posts.has_next else None
    prev_url = url_for('index', page=posts.prev_num) \
        if posts.has_prev else None
    return render_template('index.html', title='Home', form=form,
                           posts=posts.items, next_url=next_url,
                           prev_url=prev_url)

 @app.route('/explore')
 @login_required
 def explore():
    page = request.args.get('page', 1, type=int)
    posts = Post.query.order_by(Post.timestamp.desc()).paginate(
        page, app.config['POSTS_PER_PAGE'], False)
    next_url = url_for('explore', page=posts.next_num) \
        if posts.has_next else None
    prev_url = url_for('explore', page=posts.prev_num) \
        if posts.has_prev else None
    return render_template("index.html", title='Explore', posts=posts.items,
                          next_url=next_url, prev_url=prev_url)
```

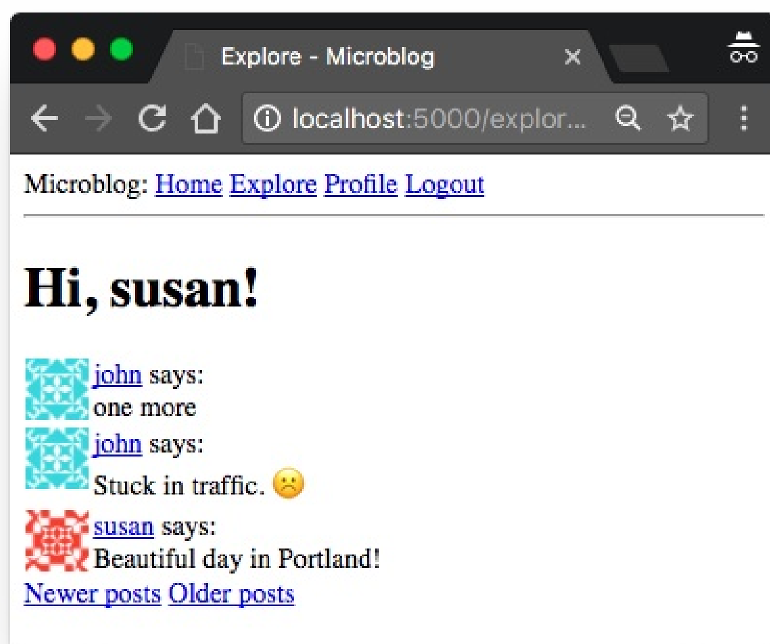
这两个视图函数中的 `next_url` 和 `prev_url` 只有在该方向上存在一个页面时，才会被设置为由 `url_for()` 返回的URL。如果当前页面位于用户动态集合的末尾或者开头，那么 `Pagination` 实例的 `has_next` 或 `has_prev` 属性将为'False'，在这种情况下，将设置该方向的链接为 `None` 。

`url_for()` 函数的一个有趣的地方是，你可以添加任何关键字参数，如果这些参数的名字没有直接在URL中匹配使用，那么Flask将它们设置为URL的查询字符串参数。

现在让我们把它们渲染在*index.html*模板上，就在用户动态列表的正下方：

```
    ...
    {% for post in posts %}
        {% include '_post.html' %}
    {% endfor %}
    {% if prev_url %}
    <a href="{{ prev_url }}">Newer posts</a>
    {% endif %}
    {% if next_url %}
    <a href="{{ next_url }}">Older posts</a>
    {% endif %}
    ...
```

主页和发现页都添加了分页链接。第一个链接标记为“Newer posts”，并指向前一页（请记住，我显示的用户动态按时间的倒序来排序，所以第一页是最新的内容）。第二个链接标记为“Older posts”，并指向下一页的帖子。如果这两个链接中的任何一个都是 `None` ，则通过条件过滤将其从页面中省略。



个人主页中的分页

主页分页已经完成，但是，个人主页中也有一个用户动态列表，其中只显示个人主页拥有者的动态。为了保持一致，个人主页也应该实现分页，以匹配主页的分页样式。

我开始更新个人主页视图函数，其中仍然有一个模拟用户动态的列表。

```
@app.route('/user/<username>')
@login_required
def user(username):
    user = User.query.filter_by(username=username).first_or_404()
    page = request.args.get('page', 1, type=int)
    posts = user.posts.order_by(Post.timestamp.desc()).paginate(
        page, app.config['POSTS_PER_PAGE'], False)
    next_url = url_for('user', username=user.username, page=posts.next_num) \
        if posts.has_next else None
    prev_url = url_for('user', username=user.username, page=posts.prev_num) \
        if posts.has_prev else None
    return render_template('user.html', user=user, posts=posts.items,
                           next_url=next_url, prev_url=prev_url)
```

为了得到用户的动态列表，我利用了`User`模型中已经定义好的`user.posts`一对多关系。 我执行该查询并添加一个`order_by（）`子句，以便我首先得到最新的用户动态，然后完全按照我对主页和发现页面中的用户动态所做的那样进行分页。 请注意，由`url_for()`函数生成的分页链接需要额外的`username`参数，因为它们指向个人主页，个人主页依赖用户名作为URL的动态组件。

最后，对*user.html*模板的更改与我在主页上所做的更改相同：

```
    ...
    {% for post in posts %}
        {% include '_post.html' %}
    {% endfor %}
    {% if prev_url %}
    <a href="{{ prev_url }}">Newer posts</a>
    {% endif %}
    {% if next_url %}
    <a href="{{ next_url }}">Older posts</a>
    {% endif %}
```

完成对分页功能的实验后，可以将`POSTS_PER_PAGE`配置项设置为更合理的值：

```
class Config(object):
    # ...
    POSTS_PER_PAGE = 25
```

本文翻译自The Flask Mega-Tutorial Part X: Email Support

这是Flask Mega-Tutorial系列的第十部分，在其中我将告诉你，应用如何向你的用户发送电子邮件，以及如何在电子邮件支持之上构建密码重置功能。

现在，应用在数据库方面做得相当不错，所以在本章中，我想抛开这个主题，开始添加发送电子邮件的功能，这是大多数Web应用必需的另一个重要部分。

为什么应用需要发送电子邮件给用户？ 原因很多，但其中一个常见的原因是解决与认证相关的问题。 在本章中，我将为忘记密码的用户添加密码重置功能。 当用户请求重置密码时，应用将发送包含特制链接的电子邮件。 用户然后需要点击该链接才能访问设置新密码的表单。

本章的*GitHub*链接为：*Browse*, *Zip*, *Diff.*

Flask-Mail简介

就实际的邮件发送而言，Flask有一个名为Flask-Mail的流行插件，可以使任务变得非常简单。 和往常一样，该插件是用pip安装的：

```
(venv) $ pip install flask-mail
```

密码重置链接将包含有一个安全令牌。 为了生成这些令牌，我将使用JSON Web Tokens，它也有一个流行的Python包：

```
(venv) $ pip install pyjwt
```

Flask-Mail插件是通过 `app.config` 对象来配置的。还记得在第七章中，我添加了用于在生产环境中发生错误时发送电子邮件的配置项? 当时我没有告诉你，不过，我选择的配置变量都是Flask-Mail的需求的，所以不需要任何额外的工作，配置的活已经完工。

像大多数Flask插件一样，你需要在Flask应用创建之后创建一个邮件实例。 本处， `mail` 是类 `Mail` 的一个实例：

```
# ...
from flask_mail import Mail

app = Flask(__name__)
# ...
mail = Mail(app)
```

第七章中我提到过，测试发送电子邮件的方式有两种。 如果你想使用一个模拟的电子邮件服务器，Python提供了一个非常好用的方法，你可以使用下面的命令在第二个终端中启动它：

```
(venv) $ python -m smtpd -n -c DebuggingServer localhost:8025
```

要配置此服务器，需要设置两个环境变量：

```
(venv) $ export MAIL_SERVER=localhost
(venv) $ export MAIL_PORT=8025
```

如果你希望真实地发送电子邮件，则需要使用真实的电子邮件服务器。 那么你只需要为它设置 `MAIL_SERVER` 、 `MAIL_PORT` 、 `MAIL_USE_TLS` 、 `MAIL_USERNAME` 和 `MAIL_PASSWORD` 环境变量。如果你想要快速解决方案，可以使用Gmail帐户发送电子邮件，并使用以下设置：

```
(venv) $ export MAIL_SERVER=smtp.googlemail.com
(venv) $ export MAIL_PORT=587
(venv) $ export MAIL_USE_TLS=1
(venv) $ export MAIL_USERNAME=<your-gmail-username>
(venv) $ export MAIL_PASSWORD=<your-gmail-password>
```

如果你使用的是Microsoft Windows，则需要在上面的每个 `export` 语句中将 `export` 替换为 `set` 。

Gmail帐户中的安全功能可能会阻止应用通过它发送电子邮件，除非你明确允许“安全性较低的应用程序”访问你的Gmail帐户。 可以阅读此处来了解具体情况，如果你担心帐户的安全性，可以创建一个辅助邮箱帐户，配置它来仅用于测试电子邮件功能，或者你可以暂时启用允许不太安全的应用程序来运行此测试，完成后恢复为默认值。

Flask-Mail的使用

为了学习Flask-Mail如何工作，我将向你展示如何用Python shell发送电子邮件。那么，运行 `flask shell` 以激活Python，然后运行下面的命令：

```
>>> from flask_mail import Message
>>> from app import mail
>>> msg = Message('test subject', sender=app.config['ADMINS'][0],
... recipients=['your-email@example.com'])
>>> msg.body = 'text body'
>>> msg.html = '<h1>HTML body</h1>'
>>> mail.send(msg)
```

上面的代码片段将发送一个电子邮件到你在 `recipients` 参数中设置的电子邮件地址列表。发件人配置项我在第七章中已经配置过了，是 `ADMINS` 。 该电子邮件将具有纯文本和HTML版本，所以根据你的电子邮件客户端的配置，可能会看到它们之中的其中之一。

如你所见，相当简单。现在让我们将电子邮件整合到应用中。

简单的电子邮件框架

我将从编写一个发送电子邮件的帮助函数开始，这个函数基本上是上一节中shell函数的通用版本。 我将把这个函数放在一个名为`app/email.py`的新模块中：

```
from flask_mail import Message
from app import mail

def send_email(subject, sender, recipients, text_body, html_body):
    msg = Message(subject, sender=sender, recipients=recipients)
    msg.body = text_body
    msg.html = html_body
    mail.send(msg)
```

Flask-Mail支持一些我不在这里使用的功能，如抄送和密件抄送列表。 如果你对这些选项感兴趣，务必查阅Flask-Mail文档。

请求重置密码

我上面提到过，用户有权利重置密码。因此我将在登录页面提供一个链接：

```
    <p>
        Forgot Your Password?
        <a href="{{ url_for('reset_password_request') }}">Click to Reset It</a>
    </p>
```

当用户点击链接时，会出现一个新的Web表单，要求用户输入注册的电子邮件地址，以启动密码重置过程。 这里是表单类：

```
class ResetPasswordRequestForm(FlaskForm):
    email = StringField('Email', validators=[DataRequired(), Email()])
    submit = SubmitField('Request Password Reset')
```

这里是相应的HTML模板：

```
{% extends "base.html" %}

{% block content %}
    <h1>Reset Password</h1>
    <form action="" method="post">
        {{ form.hidden_tag() }}
        <p>
            {{ form.email.label }}<br>
            {{ form.email(size=64) }}<br>
            {% for error in form.email.errors %}
            <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <p>{{ form.submit() }}</p>
    </form>
{% endblock %}
```

当然也需要一个视图函数来处理表单：

```
from app.forms import ResetPasswordRequestForm
from app.email import send_password_reset_email

@app.route('/reset_password_request', methods=['GET', 'POST'])
def reset_password_request():
    if current_user.is_authenticated:
        return redirect(url_for('index'))
    form = ResetPasswordRequestForm()
    if form.validate_on_submit():
        user = User.query.filter_by(email=form.email.data).first()
        if user:
            send_password_reset_email(user)
        flash('Check your email for the instructions to reset your password')
        return redirect(url_for('login'))
    return render_template('reset_password_request.html',
                           title='Reset Password', form=form)
```

该视图函数与其他的表单处理视图函数非常相似。 我从确保用户没有登录开始，如果用户登录，那么使用密码重置功能就没有意义，所以我重定向到主页。

当表格被提交并验证通过，我使用表格中的用户提供的电子邮件来查找用户。 如果我找到用户，就发送一封密码重置电子邮件。 我执行此操作使用的`send_password_reset_email()`辅助函数，将在下面向你展示。

电子邮件发送后，我会闪现一条消息，指示用户查看电子邮件以获取进一步说明，然后重定向回登录页面。 你可能会注意到，即使用户提供的电子邮件不存在，也会显示闪现的消息，这样的话，客户端就不能用这个表单来判断一个给定的用户是否已注册。

密码重置令牌

在实现`send_password_reset_email()`函数之前，我需要一种方法来生成密码重置链接，它将被通过电子邮件发送给用户 。 当链接被点击时，将为用户展现设置新密码的页面。 这个计划中棘手的部分是确保只有有效的重置链接可以用来重置帐户的密码。

生成的链接中会包含令牌，它将在允许密码变更之前被验证，以证明请求重置密码的用户是通过访问重置密码邮件中的链接而来的。JSON Web Token（JWT）是这类令牌处理的流行标准。 JWTs的优点是它是自成一体的，不但可以生成令牌，还提供对应的验证方法。

如何运行JWTs？让我们通过Python shell来学习一下：

```
>>> import jwt
>>> token = jwt.encode({'a': 'b'}, 'my-secret', algorithm='HS256')
>>> token
b'eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJhIjoiYiJ9.dvOo58OBDHiuSHD4uW88nfJikhYAXc_sfU
Hq1mDi4G0'
>>> jwt.decode(token, 'my-secret', algorithms=['HS256'])
{'a': 'b'}
```

`{'a':'b'}`字典是要写入令牌的示例有效载荷。 为了使令牌安全，需要提供一个秘密密钥用于创建加密签名。 在这个例子中，我使用了字符串`'my-secret'`，但是在应用中，我将使用配置中的`SECRET_KEY`。`algorithm`参数指定使用什么算法来生成令牌，而`HS256`是应用最广

泛的算法。

如你所见，得到的令牌是一长串字符。 但是不要认为这是一个加密的令牌。 令牌的内容，包括有效载荷，可以被任何人轻易解码（不相信我？复制上面的令牌，然后粘贴在JWT调试器上就可以看到它的内容）。 使令牌安全的是，有效载荷是被签名的。 如果有人试图伪造或篡改令牌中的有效载荷，则签名将会无效，并且生成新的签名依赖秘密密钥。 令牌验证通过时，有效负载的内容将被解码并返回给调用者。 如果令牌的签名验证通过，有效载荷才可以被认为是可信的。

我要用于密码重置令牌的有效载荷格式为 `{'reset_password'：user_id，'exp'：token_expiration}` 。 `exp` 字段是JWTs的标准，如果它存在，则表示令牌的到期时间。 如果一个令牌有一个有效的签名，但是它已经过期，那么它也将被认为是无效的。 对于密码重置功能，我会给这些令牌10分钟的有效期。

当用户点击电子邮件链接时，令牌将被作为URL的一部分发送回应用，处理这个URL的视图函数首先要做的就是验证它。 如果签名是有效的，则可以通过存储在有效载荷中的ID来识别用户。 一旦得知用户的身份，应用可以要求一个新的密码，并将其设置在用户的帐户上。

由于这些令牌属于用户，因此我将在 `User` 模型中编写令牌生成和验证的方法：

```
from time import time
import jwt
from app import app

class User(UserMixin, db.Model):
    # ...

    def get_reset_password_token(self, expires_in=600):
        return jwt.encode(
            {'reset_password': self.id, 'exp': time() + expires_in},
            app.config['SECRET_KEY'], algorithm='HS256').decode('utf-8')

    @staticmethod
    def verify_reset_password_token(token):
        try:
            id = jwt.decode(token, app.config['SECRET_KEY'],
                            algorithms=['HS256'])['reset_password']
        except:
            return
        return User.query.get(id)
```

`get_reset_password_token()` 函数以字符串形式生成一个JWT令牌。 请注意， `decode('utf-8')` 是必须的，因为 `jwt.encode()` 函数将令牌作为字节序列返回，但是在应用中将令牌表示为字符串更方便。

`verify_reset_password_token()` 是一个静态方法，这意味着它可以直接从类中调用。 静态方法与类方法类似，唯一的区别是静态方法不会接收类作为第一个参数。 这个方法需要一个令牌，并尝试通过调用PyJWT的 `jwt.decode()` 函数来解码它。 如果令牌不能被验证或已过期，将会引发异常，在这种情况下，我会捕获它以防止出现错误，然后将 `None` 返回给调用者。 如果令牌有效，那么来自令牌有效负载的 `reset_password` 的值就是用户的ID，所以我可以加载用户并返回它。

发送密码重置电子邮件

现在我有了令牌，可以生成密码重置电子邮件。 `send_password_reset_email()` 函数依赖于上面写的 `send_email()` 函数。

```
from flask import render_template
from app import app

# ...

def send_password_reset_email(user):
    token = user.get_reset_password_token()
    send_email('[Microblog] Reset Your Password',
               sender=app.config['ADMINS'][0],
               recipients=[user.email],
               text_body=render_template('email/reset_password.txt',
                                         user=user, token=token),
               html_body=render_template('email/reset_password.html',
                                         user=user, token=token))
```

这个函数中有趣的部分是电子邮件的文本和HTML内容是使用熟悉的 `render_template()` 函数从模板生成的。 模板接收用户和令牌作为参数，以便可以生成个性化的电子邮件消息。 以下是重置密码电子邮件的文本模板：

```
Dear {{ user.username }},

To reset your password click on the following link:

{{ url_for('reset_password', token=token, _external=True) }}

If you have not requested a password reset simply ignore this message.

Sincerely,

The Microblog Team
```

这是更美观的的HTML版本：

```
<p>Dear {{ user.username }},</p>
<p>
    To reset your password
    <a href="{{ url_for('reset_password', token=token, _external=True) }}">
        click here
    </a>.
</p>
<p>Alternatively, you can paste the following link in your browser's address bar:</p>
<p>{{ url_for('reset_password', token=token, _external=True) }}</p>
<p>If you have not requested a password reset simply ignore this message.</p>
<p>Sincerely,</p>
<p>The Microblog Team</p>
```

请注意，这两个电子邮件模板中的 `url_for()` 调用中引用的 `reset_password` 路由尚不存在，这将在下一节中添加。

重置用户密码

当用户点击电子邮件链接时，会触发与此功能相关的第二个路由。 这是密码重置视图函数：

```
from app.forms import ResetPasswordForm

@app.route('/reset_password/<token>', methods=['GET', 'POST'])
def reset_password(token):
    if current_user.is_authenticated:
        return redirect(url_for('index'))
    user = User.verify_reset_password_token(token)
    if not user:
        return redirect(url_for('index'))
    form = ResetPasswordForm()
    if form.validate_on_submit():
        user.set_password(form.password.data)
        db.session.commit()
        flash('Your password has been reset.')
        return redirect(url_for('login'))
    return render_template('reset_password.html', form=form)
```

在这个视图函数中，我首先确保用户没有登录，然后通过调用`User`类的令牌验证方法来确定用户是谁。 如果令牌有效，则此方法返回用户；如果不是，则返回`None`，并将重定向到主页。

如果令牌是有效的，那么我向用户呈现第二个表单，需要用户其中输入新密码。 这个表单的处理方式与以前的表单类似，表单提交验证通过后，我调用`User`类的`set_password()`方法来更改密码，然后重定向到登录页面，以便用户登录。

这是`ResetPasswordForm`类：

```
class ResetPasswordForm(FlaskForm):
    password = PasswordField('Password', validators=[DataRequired()])
    password2 = PasswordField(
        'Repeat Password', validators=[DataRequired(), EqualTo('password')])
    submit = SubmitField('Request Password Reset')
```

这是相应的HTML模板：

```
{% extends "base.html" %}

{% block content %}
    <h1>Reset Your Password</h1>
    <form action="" method="post">
        {{ form.hidden_tag() }}
        <p>
            {{ form.password.label }}<br>
            {{ form.password(size=32) }}<br>
            {% for error in form.password.errors %}
            <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <p>
            {{ form.password2.label }}<br>
            {{ form.password2(size=32) }}<br>
            {% for error in form.password2.errors %}
            <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <p>{{ form.submit() }}</p>
    </form>
{% endblock %}
```

密码重置功能现已完成，一定要多尝试几次。

异步电子邮件

如果你正在使用Python提供的模拟电子邮件服务器，可能没有注意到这一点，那就是发送电子邮件会大大减慢应用的速度，原因是发送电子邮件时所发生的和电子邮件服务器的网络交互。通常需要几秒钟的时间才能收到电子邮件，如果收件人的电子邮件服务器速度较慢，或者收件人有多个，则可能会更久。

我真正想要的`send_email()`函数是异步的。 那是什么意思？ 这意味着当这个函数被调用时，发送邮件的任务被安排在后台进行，释放`send_email()`函数以立即返回，以便应用可以在发送邮件的同时继续运行。

Python实际上有多种方式支持运行异步任务，`threading`和`multiprocessing`模块都可以做到这一点。 为发送电子邮件启动一个后台线程，比开始一个全新的进程需要的资源少得多，所以我打算采用这种方法：

```
from threading import Thread
# ...

def send_async_email(app, msg):
    with app.app_context():
        mail.send(msg)

def send_email(subject, sender, recipients, text_body, html_body):
    msg = Message(subject, sender=sender, recipients=recipients)
    msg.body = text_body
    msg.html = html_body
    Thread(target=send_async_email, args=(app, msg)).start()
```

`send_async_email` 函数现在运行在后台线程中，它通过 `send_email()` 的最后一行中的 `Thread()` 类来调用。 有了这个改变，电子邮件的发送将在线程中运行，并且当进程完成时，线程将结束并自行清理。 如果你已经配置了一个真正的电子邮件服务器，当你按下密码重置请求表单上的提交按钮时，肯定会注意到访问速度的提升。

你可能预期只有 `msg` 参数会被发送到线程，但正如你在代码中所看到的那样，我也传入了应用实例。 使用线程时，需要牢记Flask的一个重要设计方面。 Flask使用上下文来避免必须跨函数传递参数。 我不打算详细讨论这个问题，但是需要知道的是，有两种类型的上下文，即应用上下文和请求上下文。 在大多数情况下，这些上下文由框架自动管理，但是当应用启动自定义线程时，可能需要手动创建这些线程的上下文。

许多Flask插件需要应用上下文才能工作，因为这使得他们可以在不传递参数的情况下找到Flask应用实例。这些插件需要知道应用实例的原因是因为它们的配置存储在 `app.config` 对象中，这正是Flask-Mail的情况。 `mail.send()` 方法需要访问电子邮件服务器的配置值，而这必须通过访问应用属性的方式来实现。 使用 `with app.app_context()` 调用创建的应用上下文使得应用实例可以通过来自Flask的 `current_app` 变量来进行访问。

本文翻译自The Flask Mega-Tutorial Part XI: Facelift

这是Flask Mega-Tutorial系列的第十一部分，我将告诉你如何用基于Bootstrap用户界面框架的新模板替换基础的HTML模板。

你把玩Microblog应用也有一段时间了，所以我相信你已经注意到，我没有花太多时间来美化它，说得更具体点，我根本没有花时间。 所有的模板只使用了基础样式，没有任何自定义的展现。 这对于我来说却非常有用，因为我可以专注于应用的实际逻辑，不用分心于编写好看的HTML和CSS代码。

但是我已经长期关注应用的后端部分一段时间了。 因此在本章中，我暂停一下后端的工作，并花点时间向你展示如何使应用看起来更加优雅和专业。

本章将与之前的章节略有不同，因为我不会像平常解说Python那样，事无巨细，一一道来，毕竟Python才是本教程的主要内容。 创建漂亮的网页是一个很广泛的话题，而与Python Web的后端开发很大程度上无关，因此我将讨论一些基本的指导方针和想法，你可以通过重新设计应用的外观来研究和学习它。

本章的*GitHub*链接为：*Browse*, *Zip*, *Diff.*

CSS框架

虽然我们可以争辩说写代码不容易，但是与那些必须让网页在所有Web浏览器上具有良好一致外观的网页设计师相比，我们的痛苦不值一提。 虽然近年来这种情况得到一定程度的缓解，但是在一些浏览器中仍然存在着晦涩的错误或奇怪的设定，这使得设计网页的任务变得非常困难。 如果还需要兼容屏幕限制设备（诸如平板电脑和智能手机）的浏览器，则更加困难。

如果你和我一样，只是一个想创建出规范网页的开发人员，没有时间或兴趣去学习底层机制并通过编写原生HTML和CSS来实现它，那么唯一可行的解决方案是使用*CSS*框架来简化任务。 通过这种方式，你会失去一些创造性的自由，但另一方面，无需通过太多的功夫就可以让网页在所有浏览器中看起来都不错。 CSS框架为普通类型的用户界面元素提供了高级CSS类的集合，其中包含预定义样式。 大多数这样的框架还提供JavaScript插件，以实现不能纯粹使用HTML和CSS来完成的功能。

Bootstrap简介

最受欢迎的CSS框架之一是由Twitter推出的Bootstrap。 如果你想看看这个框架可以设计的页面类型，文档有一些示例。

这些是使用Bootstrap来设置网页风格的一些好处：

- 在所有主流网页浏览器中都有相似的外观

- 自动处理PC桌面，平板电脑和手机屏幕尺寸
- 可定制的布局
- 精心设计的导航栏，表单，按钮，警示，弹出窗口等

使用Bootstrap最直接的方法是简单地在你的基本模板中导入*bootstrap.min.css*文件。 可以下载此文件并将其添加到你的项目中，或直接从CDN导入。 然后，你可以根据其文档开始使用它提供的通用CSS类，实在是太棒了。 你可能还需要导入包含框架JavaScript代码的*bootstrap.min.js*文件，以便使用最先进的功能。

幸运的是，有一个名为Flask-Bootstrap的Flask插件，它提供了一个已准备好的基础模板，该模板引入了Bootstrap框架。 让我们来安装这个扩展：

```
(venv) $ pip install flask-bootstrap
```

使用Flask-Bootstrap

Flask-Bootstrap需要像大多数其他Flask插件一样被初始化：

*app/**init**.py*: Flask-Bootstrap实例。

```
# ...
from flask_bootstrap import Bootstrap

app = Flask(__name__)
# ...
bootstrap = Bootstrap(app)
```

在初始化插件之后，*bootstrap/base.html*模板就会变为可用状态，你可以使用 `extends` 子句从应用模板中引用。

但是，回顾一下，我已经使用了 `extends` 子句来继承我的基础模板，这使我可以将页面的公共部分放在一个地方。 *base.html*模板定义了导航栏，其中包含几个链接，并且还导出了一个 `content` 块。 应用中的所有其他模板都从基础模板继承，并为内容块提供页面的主要内容。

那么我怎样才能适配Bootstrap基础模板呢？ 解决方案是从使用两个层级到使用三个层级。 *bootstrap/base.html*模板提供页面的基本结构，其中引入了Bootstrap框架文件。 这个模板为派生的模板定义了一些块，例如 `title` ， `navbar` 和 `content` （参见块的完整列表）。 我将更改*base.html*模板以从*bootstrap/base.html*派生，并提供 `title` ， `navbar` 和 `content` 块的实现。 反过来，*base.html*将为从其派生的模板导出 `app_content` 块以定义页面内容。

下面你可以看到从Bootstrap基础模板派生的*base.html*的代码。 请注意，此列表不包含导航栏的整个HTML，但你可以在GitHub上或下载本章的代码来查看完整的实现。

app/templates/base.html：重新设计后的基础模板。

```
{% extends 'bootstrap/base.html' %}

{% block title %}
    {% if title %}{{ title }} - Microblog{% else %}Welcome to Microblog{% endif %}
{% endblock %}

{% block navbar %}
    <nav class="navbar navbar-default">
        ... navigation bar here (see complete code on GitHub) ...
    </nav>
{% endblock %}

{% block content %}
    <div class="container">
        {% with messages = get_flashed_messages() %}
        {% if messages %}
            {% for message in messages %}
            <div class="alert alert-info" role="alert">{{ message }}</div>
            {% endfor %}
        {% endif %}
        {% endwith %}

        {# application content needs to be provided in the app_content block #}
        {% block app_content %}{% endblock %}
    </div>
{% endblock %}
```

从中你可以看到我如何从*bootstrap/base.html*派生此模板，接下来分别实现了页面标题、导航栏和页面内容的这三个模块。

`title` 块需要使用 `<title>` 标签来定义用于页面标题的文本。 对于这个块我简单地挪用了原始基本模板中 `<title>` 标签内部的逻辑。

`navbar` 块是一个可选块，用于定义导航栏。 对于此块，我调整了Bootstrap导航栏文档中的示例，以便它在左侧展示网站品牌，跟着是Home和Explore的链接。 然后我添加了个人主页和登录或注销链接并使其与页面的右边界对齐。 正如我上面提到的，我在上面的例子中省略了HTML，但是你可以从本章的下载包中获得完整的*base.html*模板。

最后，在 `content` 块中，我定义了一个顶级容器，并在其中设定了呈现闪现消息的逻辑，这些消息现在将显示为Bootstrap警示的样式。 接下来是一个新的 `app_content` 块，这个块用于从其派生的模板来定义他们自己的内容。

所有页面模板的原始版本在名为 `content` 的块中定义了它们的内容。 正如你在上面看到的，Flask-Bootstrap使用名为 `content` 的块，所以我将我的内容块重命名为 `app_content` 。 所以我所有的模板都必须重命名为使用 `app_content` 作为它们的内容块。 例如，这是*404.html*模板的修改后版本的展示：

app/templates/404.html：重新设计后的404错误模板。

```
{% extends "base.html" %}

{% block app_content %}
    <h1>File Not Found</h1>
    <p><a href="{{ url_for('index') }}">Back</a></p>
{% endblock %}
```

渲染Bootstrap表单

Flask-Bootstrap在渲染表单这方面做得非常出色。 Flask-Bootstrap不需要逐个设置表单字段，而是使用一个接受Flask-WTF表单对象作为参数的宏，并以Bootstrap样式渲染出完整的表单。

下面你可以看到重新设计后的*register.html*模板：

app/templates/register.html: ：用户注册模板。

```
{% extends "base.html" %}
{% import 'bootstrap/wtf.html' as wtf %}

{% block app_content %}
    <h1>Register</h1>
    <div class="row">
        <div class="col-md-4">
            {{ wtf.quick_form(form) }}
        </div>
    </div>
{% endblock %}
```

是不是很棒？ 顶端附近的 `import` 语句与Python导入类似。 这增加了一个 `wtf.quick_form()` 宏，它在单行代码中渲染完整的表单，包括对显示验证错误的支持，并且适配Bootstrap框架的所有样式。

再一次地，我不会向你展示我为应用中的其他表单所做的所有更改，但这些更改都是可以在GitHub上下载或检查到的。

渲染用户动态

单条用户动态的渲染逻辑被提取到名为*_post.html*的子模板中。 我只需要在这个模板上做一些很小的调整，就可以使其在Bootstrap下看起来很棒了。

app/templates/_post.html: 重新设计后的用户动态子模板。

```
    <table class="table table-hover">
        <tr>
            <td width="70px">
                <a href="{{ url_for('user', username=post.author.username) }}">
                    <img src="{{ post.author.avatar(70) }}" />
                </a>
            </td>
            <td>
                <a href="{{ url_for('user', username=post.author.username) }}">
                    {{ post.author.username }}
                </a>
                says:
                <br>
                {{ post.body }}
            </td>
        </tr>
    </table>
```

渲染分页链接

分页链接是Bootstrap提供直接支持的另一个方面。 为此，我再一次访问Bootstrap 文档，并修改了其中的一个示例。 以下是在*index.html*页面中的分页链接的代码：

app/templates/index.html: 重新设计后的分页链接。

```
    ...
    <nav aria-label="...">
        <ul class="pager">
            <li class="previous{% if not prev_url %} disabled{% endif %}">
                <a href="{{ prev_url or '#' }}">
                    <span aria-hidden="true">&larr;</span> Newer posts
                </a>
            </li>
            <li class="next{% if not next_url %} disabled{% endif %}">
                <a href="{{ next_url or '#' }}">
                    Older posts <span aria-hidden="true">&rarr;</span>
                </a>
            </li>
        </ul>
    </nav>
```

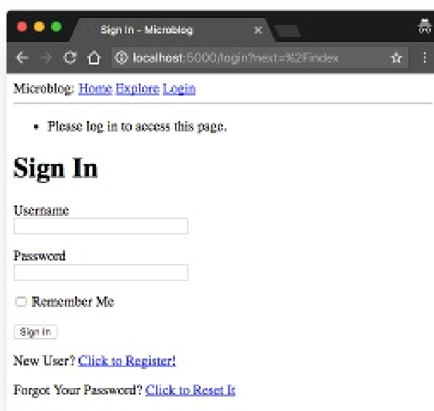
请注意，在分页链接的实现中，当某个方向没有更多内容时，不是隐藏该链接，而是使用禁用状态，这会使该链接显示为灰色。

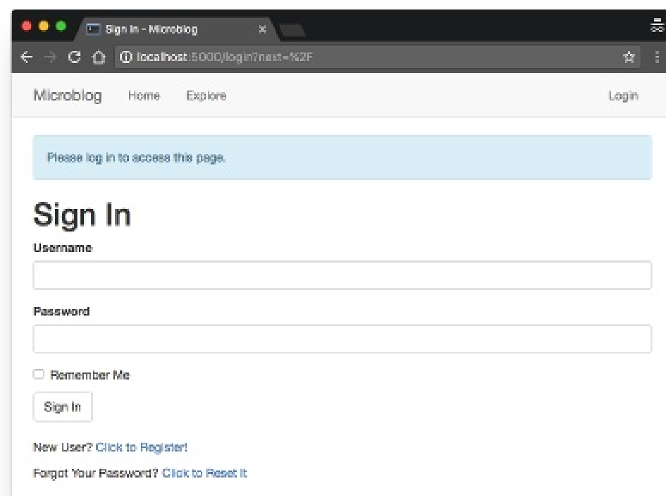
类似的更改需要应用于*user.html*，但我不打算展示在此处。 本章的下载包中包含这些更改。

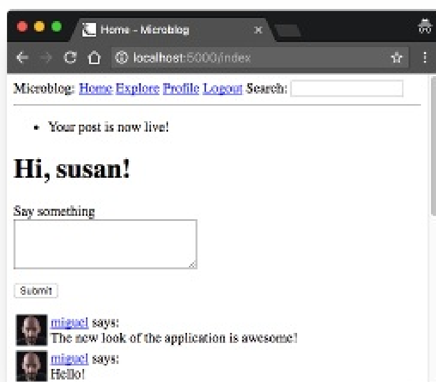
对比

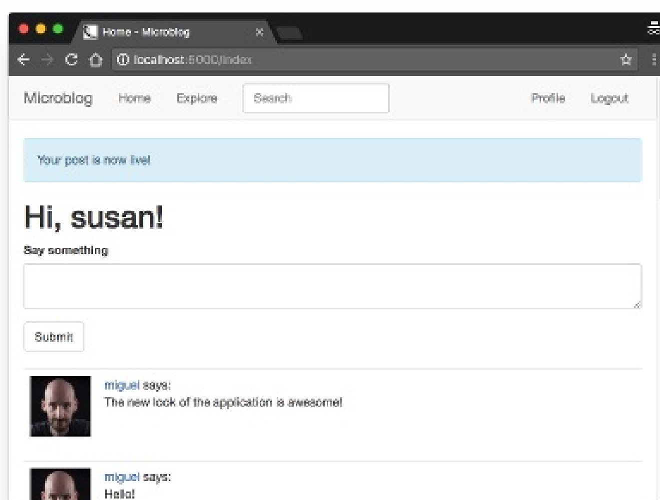
请下载本章的zip文件并更新应用。

下面你可以对照几张美化前后的图片来观察转变情况。 请记住，这种转变是在不改变一行应用逻辑代码的情况下实现的！






Home - Microblog
localhost:5000/index
Microblog: Home Explore Profile Logout Search:
Your post is now live!
Hi, susan!
Say something
Submit
miguel says:
The new look of the application is awesome!
miguel says:
Hello!


Home - Microblog
localhost:5000/index
Microblog
Home
Explore
Search
Profile
Logout
Your post is now live!
Hi, susan!
Say something
Submit
miguel says:
The new look of the application is awesome!
miguel says:
Hello!

本文翻译自The Flask Mega-Tutorial Part XII: Dates and Times

这是Flask Mega-Tutorial系列的第十二部分，我将告诉你如何以适配所有用户的方式处理日期和时间，无论他们身处地球上的何处。

显示日期和时间是Microblog应用中长期被忽略的其中一个方面。 直到现在，我也只是让Python渲染了`User`模型中的`datetime`对象，并且完全忽略了`Post`模型中的`datetime`对象。

本章的*GitHub*链接为：*Browse*, *Zip*, *Diff.*

时区地狱

使用服务器端的Python渲染日期和时间来展示到用户的浏览器并非一个好主意。考虑如下的例子， 我在2017年9月28日下午4点06分写这篇文章。我身处的时区是PDT(UTC-7)，在Python解释器中运行如下：

```
>>> from datetime import datetime
>>> str(datetime.now())
'2017-09-28 16:06:30.439388'
>>> str(datetime.utcnow())
'2017-09-28 23:06:51.406499'
```

`datetime.now()`调用返回我所处位置的本地时间，而`datetime.utcnow()`调用则返回UTC时区中的时间。 如果我可以让遍布世界不同地区的多人同时运行上面的代码，那么`datetime.now()`函数将为他们每个人返回不同的结果，但是无论位置如何，`datetime.utcnow()`总是会返回同一时间。 那么你认为哪一个更适合用在一个很可能其用户遍布世界各地的Web应用中呢？

很明显，服务器必须管理一致且独立于位置的时间。 如果这个应用增长到在全世界不同地区都需要部署生产服务器的时候，我不希望每个服务器都在写入不同时区的时间戳到数据库，因为这会导致其无法正常地运行。 由于UTC是最常用的统一时区，并且在`datetime`类中也受到支持，因此我将会使用它。

但这种方法存在一个严重问题。 对处于不同时区的用户，如果他们看到的是UTC时区中的时间，那么很难确定是何时发布的信息。 他们需要事先知道展示的时间是UTC时区的，才能在精神上调整自己的时区。 设想一下PDT时区中的一个用户在下午3点发布了一些内容，并立即看到该帖子以UTC时间表示的晚上10:00或更准确的22:00，这太混乱了。

从服务器的角度来说，将时间戳标准化为UTC，意义重大，但这会为用户带来可用性问题。 本章的目标就是解决该问题，同时保持服务器中以UTC格式管理的所有时间戳。 While standardizing the timestamps to UTC makes a lot of sense from the server's perspective, this creates a usability problem for users. The goal of this chapter is to address this problem while keeping all the timestamps managed in the server in UTC.

时区转换

该问题的直接解决方案是将所有时间戳从存储的UTC单位转换为每个用户的本地时间。 这样一来，服务器可以继续使用UTC来保持时区的一致性，而针对每个用户量身定制的即时转换来解决可用性问题。 这个解决方案棘手的部分是要知道每个用户的位置。

许多网站都有一个配置页面供用户指定他们的时区。 这将需要我添加一个新的页面，其中我向用户显示带有时区列表的下拉列表。 也可能用户在第一次访问网站时，作为注册的一部分，会被要求输入他们的时区。

虽然该方案可以解决问题，但要求用户输入他们已经在其操作系统中配置的信息有点奇怪。如果我能从他们的计算机中获取时区设置，似乎效率会更高。

事实证明，Web浏览器可以获取用户的时区，并通过标准的日期和时间JavaScript API暴露它。 实际上有两种方法来利用JavaScript提供的时区信息：

- “老派”方法是当用户第一次登录到应用程序时，Web浏览器以某种方式将时区信息发送到服务器。 这可以通过Ajax)调用完成，或者更简单地使用meta refresh tag。 一旦服务器知道了时区，就可以将其保存在用户的会话中，或者将其写入用户在数据库中的条目中，然后在渲染模板时从中调整所有时间戳。
- “新派”的做法是不改变服务器中的东西，而在客户端中使用JavaScript来对UTC和本地时区之间进行转换。

两种选择都是有效的，但第二种选择有很大优势。 光是知道用户的时区并不足以以用户期望的格式呈现日期和时间。 浏览器还可以访问系统区域配置，该配置指定AM/PM与24小时制，DD/MM/YYYY与MM/DD/YYYY，以及许多其他文化或地区风格之类的东西。

如果这还不够，新派方法还有另一个优势，用一个开源的库来完成所有这些工作！

Moment.js和Flask-Moment简介

Moment.js是一个小型的JavaScript开源库，它将日期和时间转换成目前可以想象到的所有格式。 不久前，我创建了Flask-Moment，一个小型Flask插件，它可以使你在应用中轻松使用moment.js。

因此，让我们从安装Flask-Moment来开始吧：

```
(venv) $ pip install flask-moment
```

使用常规方法添加该插件到Flask应用中：

`app/__init__.py`：Flask-Moment实例。

```
# ...
from flask_moment import Moment

app = Flask(__name__)
# ...
moment = Moment(app)
```

与其他插件不同的是，Flask-Moment与*moment.js*一起工作，因此应用的所有模板都必须包含moment.js。为了确保该库始终可用，我将把它添加到基础模板中，可以通过两种方式完成。 最直接的方法是显式添加一个`<script>`标签来引入库，但Flask-Moment的`moment.include_moment()`函数可以更容易地实现它，它直接生成了一个`<script>`标签并在其中包含moment.js：

app/templates/base.html：在基础模板中包含moment.js

```
...

{% block scripts %}
    {{ super() }}
    {{ moment.include_moment() }}
{% endblock %}
```

我在这里添加的`scripts`块是Flask-Bootstrap基础模板暴露的另一个块，这是JavaScript引入的地方。该块与之前的块不同的地方在于它已经在基础模板中定义了一些内容了。我想要追加moment.js库的话，就需要使用`super()`语句，才能继承基础模板中已有的内容，否则就是替换。

使用Moment.js

Moment.js为浏览器提供了一个`moment`类。 呈现时间戳的第一步是创建此类的对象，并以ISO 8601格式传递所需的时间戳。 这里是一个例子：

```
t = moment('2017-09-28T21:45:23Z')
```

如果你对日期和时间不熟悉ISO 8601标准格式，格式如下：`{{ year }}-{{ month }}-{{ day }}T{{ hour }}:{{ minute }}:{{ second }}{{ timezone }}`。我已经决定我只使用UTC时区，因此最后一部分总是将会是`Z`，它表示ISO 8601标准中的UTC。

`moment`对象为不同的渲染选项提供了几种方法。 以下是一些最常见的几种：

```
moment('2017-09-28T21:45:23Z').format('L')
"09/28/2017"
moment('2017-09-28T21:45:23Z').format('LL')
"September 28, 2017"
moment('2017-09-28T21:45:23Z').format('LLL')
"September 28, 2017 2:45 PM"
moment('2017-09-28T21:45:23Z').format('LLLL')
"Thursday, September 28, 2017 2:45 PM"
moment('2017-09-28T21:45:23Z').format('dddd')
"Thursday"
moment('2017-09-28T21:45:23Z').fromNow()
"7 hours ago"
moment('2017-09-28T21:45:23Z').calendar()
"Today at 2:45 PM"
```

此示例创建了一个moment对象，该对象被初始化为2017年9月28日晚上9:45 UTC。 你可以看到，我上面尝试的所有选项都以UTC-7时区来呈现，因为这是我计算机上配置的时区。你可以在microblog上进行此操作，只要你引入了moment.js。或者你也可以在*https://momentjs.com/*上尝试。

请注意不同的方法是如何创建的不同的表示。 使用 `format()` ，你可以控制字符串的输出格式，类似于Python中的strftime函数。 `fromNow()` 和 `calendar()` 方法很有趣，因为它们会根据当前时间显示时间戳，因此你可以获得诸如"一分钟前"或"两小时内"等输出。

如果你直接在JavaScript中运行，则上述调用将返回渲染后的时间戳字符串。 然后，你可以将此文本插入页面上的适当位置，不幸的是，这需要JavaScript与DOM配合使用。 Flask-Moment插件通过启用一个类似于JavaScript上的 `moment` 对象，大大简化了对moment.js的使用，并融合了所需的JavaScript逻辑，使渲染后的时间展示在页面上。

我们来看看出现在个人主页中的时间戳。 当前的*user.html*模板使用Python生成时间的字符串表示。 现在我可以使用Flask-Moment渲染此时间戳，如下所示：

app/templates/user.html: 使用moment.js渲染时间戳。

```
                {% if user.last_seen %}
                <p>Last seen on: {{ moment(user.last_seen).format('LLL') }}</p>
                {% endif %}
```

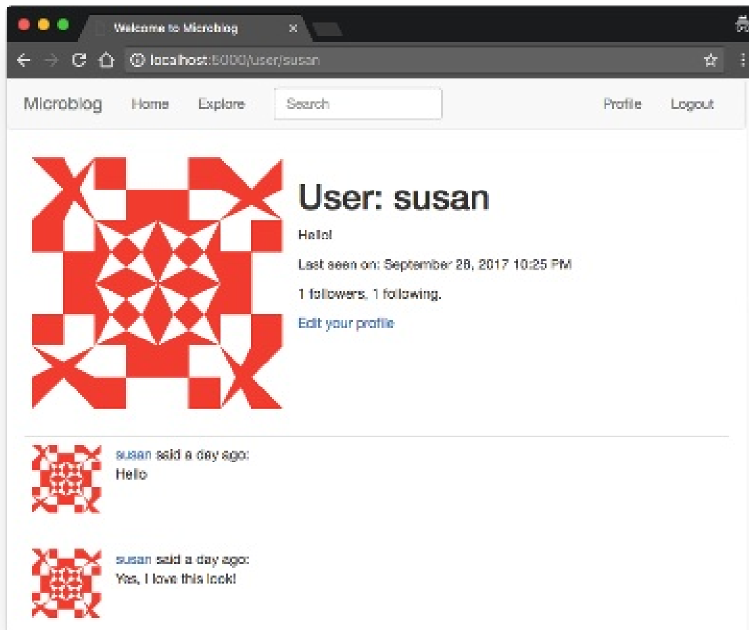
如你所见，Flask-Moment使用的语法类似于JavaScript库的语法，其中一个区别是， `moment()` 的参数现在是Python的 `datetime` 对象，而不是ISO 8601字符串。 从模板发出的 `moment()` 调用也会自动生成所需的JavaScript代码，以将呈现的时间戳插入DOM的适当位置。

我可以利用Flask-Moment和moment.js的第二个地方是被主页和个人主页调用的*_post.html*子模板。 在该模板的当前版本中，每条用户动态都以"用户名说："行开头。 现在我可以添加一个用 `fromNow()` 渲染的时间戳：

app/templates/_post.html: 在用户动态子模板中渲染时间戳。

```
                    <a href="{{ url_for('user', username=post.author.username) }}">
                        {{ post.author.username }}
                    </a>
                    said {{ moment(post.timestamp).fromNow() }}:
                    <br>
                    {{ post.body }}
```

下面，你可以看到这两个时间戳在Flask-Moment和moment.js的渲染下，表现如何：



本文翻译自The Flask Mega-Tutorial Part XIII: I18n and L10n

这是Flask Mega-Tutorial系列的第十三部分，我将告诉你如何扩展Microblog应用以支持多种语言。 作为其中的一部分，你还将学习如何为flask命令创建自己的CLI扩展。

本章的主题是国际化和本地化，通常缩写为I18n和L10n。 为了使我的应用对不会英语的人更加友好，我将在语言翻译机制的帮助下，实施翻译工作流程，来使用多种语言向用户提供服务。

本章的GitHub链接为：*Browse*, *Zip*, *Diff*.

Flask-Babel简介

你猜对了，Flask-Babel正是用于简化翻译工作的。可以使用pip命令安装它：

```
(venv) $ pip install flask-babel
```

Flask-Babel的初始化与之前的插件类似：

`app/__init__.py`：Flask-Babel实例。

```
# ...
from flask_babel import Babel

app = Flask(__name__)
# ...
babel = Babel(app)
```

作为本章的一部分，我将向你展示如何将应用翻译成西班牙语，因为我碰巧会这种语言。 我当然也可以与翻译机制合作来支持其他语言。 为了跟踪支持的语言列表，我将添加一个配置变量：

config.py：支持的语言列表。

```
class Config(object):
    # ...
    LANGUAGES = ['en', 'es']
```

我为本应用使用双字母代码来表示语言种类，但如果你需要更具体，还可以添加国家代码。 例如，你可以使用`en-US`，`en-GB`和`en-CA`来支持美国、英国和加拿大的英语以示区分。

`Babel`实例提供了一个`localeselector`装饰器。 为每个请求调用装饰器函数以选择用于该请求的语言：

`app/__init__.py`：选择最匹配的语言。

```
from flask import request

# ...

@babel.localeselector
def get_locale():
    return request.accept_languages.best_match(app.config['LANGUAGES'])
```

这里我使用了Flask中 `request` 对象的属性 `accept_languages` 。 `request` 对象提供了一个高级接口，用于处理客户端发送的带Accept-Language头部的请求。 该头部指定了客户端语言和区域设置首选项。 该头部的内容可以在浏览器的首选项页面中配置，默认情况下通常从计算机操作系统的语言设置中导入。 大多数人甚至不知道存在这样的设置，但是这是有用的，因为应用可以根据每个语言的权重，提供优选语言的列表。 为了满足你的好奇心，下面是一个复杂的 `Accept-Languages` 头部的例子：

```
Accept-Language: da, en-gb;q=0.8, en;q=0.7
```

这表示丹麦语（ `da` ）是首选语言（默认权重= 1.0），其次是英式英语（ `en-GB` ），其权重为0.8，最后是通用英语（ `en` ），权重为0.7。

要选择最佳语言，你需要将客户请求的语言列表与应用支持的语言进行比较，并使用客户端提供的权重，查找最佳语言。 这样做的逻辑有点复杂，但它已经全部封装在 `best_match()` 方法中了，该方法将应用提供的语言列表作为参数并返回最佳选择。

标记文本以在Python源代码中执行翻译

好吧，坏消息来了。 支持多语言的常规流程是在源代码中标记所有需要翻译的文本。 文本标记后，Flask-Babel将扫描所有文件，并使用gettext工具将这些文本提取到单独的翻译文件中。 不幸的是，这是一个繁琐的任务，并且是启用翻译的必要条件。

我将在这里向你展示标记操作的几个示例，你也可以从下载包获取本章完整的更改集，当然，也可以直接查看GitHub的页面。

为翻译而标记文本的方式是将它们封装在一个函数调用中，该函数调用为 `_()` ，仅仅是一个下划线。最简单的情况是源代码中出现的字符串。下面是一个 `flask()` 语句的例子：

```
from flask_babel import _
# ...
flash(_('Your post is now live!'))
```

`_()` 函数用于原始语言文本（在这种情况下是英文）的封装。 该函数将使用由 `localeselector` 装饰器装饰的选择函数，来为给定客户端查找正确的翻译语言。 `_()` 函数随后返回翻译后的文本，在本处，翻译后的文本将成为 `flash()` 的参数。

但是不可能每个情况都这么简单，试想如下的另一个 `flash()` 调用：

```
flash('User {} not found.'.format(username))
```

该文本具有一个安插在静态文本中间的动态组件。 `_()` 函数的语法支持这种类型的文本，但它基于旧版本的字符串替换语法：

```
flash(_('User %(username)s not found.', username=username))
```

还有更难处理的情况。 有些字符串文字并非是在发生请求时分配的，比如在应用启动时。因此在评估这些文本时，无法知道要使用哪种语言。 一个例子是与表单字段相关的标签，处理这些文本的唯一解决方案是找到一种方法来延迟对字符串的评估，直到它被使用，比如有实际上的请求发生了。 Flask-Babel提供了一个称为 `lazy_gettext()` 的 `_()` 函数的延迟评估的版本：

```
from flask_babel import lazy_gettext as _l

class LoginForm(FlaskForm):
    username = StringField(_l('Username'), validators=[DataRequired()])
    # ...
```

在这里，我正在导入的这个翻译函数被重命名为 `_l()` ，以使其看起来与原始的 `_()` 相似。 这个新函数将文本包装在一个特殊的对象中，这个对象会在稍后的字符串使用时触发翻译。

Flask-Login插件只要将用户重定向到登录页面，就会闪现消息。 此消息为英文，来自插件本身。 为了确保这个消息也能被翻译，我将重写默认消息，并用 `_l()` 函数进行延迟处理：

```
login = LoginManager(app)
login.login_view = 'login'
login.login_message = _l('Please log in to access this page.')
```

标记文本以在模板中进行翻译

在前面的章节中，你已经看到了如何在Python源代码中标记可翻译的文本，但这只是该过程的一部分，因为模板文件也包含文本。 `_()` 函数也可以在模板中使用，所以过程非常相似。例如，参考来自*404.html*的这段HTML代码：

```
<h1>File Not Found</h1>
```

启用翻译之后的版本是：

```
<h1>{{ _('File Not Found') }}</h1>
```

请注意，除了用 `_()` 包装文本外，还需要添加 `{{...}}` 来强制 `_()` 进行翻译，而不是将其视为模板中的文本字面量。

对于具有动态组件的更复杂的短语，也可以使用参数：

```
<h1>{{ _('Hi, %(username)s!', username=current_user.username) }}</h1>
```

*_post.html*中的一个特别棘手的案例让我花了一些时间才理顺：

```
        {% set user_link %}
            <a href="{{ url_for('user', username=post.author.username) }}">
                {{ post.author.username }}
            </a>
        {% endset %}
        {{ _('%(username)s said %(when)s',
            username=user_link, when=moment(post.timestamp).fromNow()) }}
```

这里的问题是我希望`username`是一个超链接，指向用户的个人主页，而不仅仅是名字，所以我必须使用`set`和`endset`模板指令创建一个名为`user_link`的中间变量 ，然后将其作为参数传递给翻译函数。

正如我上面提到的，你可以下载该版本的应用，其中的Python源代码和模板中都已被标记成可翻译文本。

提取文本进行翻译

一旦应用所有`_()`和`_l()`都到位了，你可以使用`pybabel`命令将它们提取到一个*.pot*文件中，该文件代表可移植对象模板。 这是一个文本文件，其中包含所有标记为需要翻译的文本。 这个文件的目的是作为一个模板来为每种语言创建翻译文件。

提取过程需要一个小型配置文件，告诉pybabel哪些文件应该被扫描以获得可翻译的文本。 下面你可以看到我为这个应用创建的*babel.cfg*：

babel.cfg：PyBabel配置文件。

```
[python: app/**.py]
[jinja2: app/templates/**.html]
extensions=jinja2.ext.autoescape,jinja2.ext.with_
```

前两行分别定义了Python和Jinja2模板文件的文件名匹配模式。 第三行定义了Jinja2模板引擎提供的两个扩展，以帮助Flask-Babel正确解析模板文件。

可以使用以下命令来将所有文本提取到 *.pot* 文件：

```
(venv) $ pybabel extract -F babel.cfg -k _l -o messages.pot .
```

`pybabel extract`命令读取`-F`选项中给出的配置文件，然后从命令给出的目录（当前目录或本处的`.`）扫描与配置的源匹配的目录中的所有代码和模板文件。 默认情况下，`pybabel`将查找`_()`以作为文本标记，但我也使用了重命名为`_l()`的延迟版本，所以我需要用`-k _l`来告诉该工具也要查找它 。 `-o`选项提供输出文件的名称。

我应该注意，*messages.pot*文件不需要合并到项目中。 这是一个只要再次运行上面的命令，就可以在需要时轻松地重新生成的文件。 因此，不需要将该文件提交到源代码管理。

生成语言目录

该过程的下一步是在除了原始语言（在本例中为英语）之外，为每种语言创建一份翻译。 我要从添加西班牙语（语言代码`es`）开始，所以这样做的命令是：

```
(venv) $ pybabel init -i messages.pot -d app/translations -l es
creating catalog app/translations/es/LC_MESSAGES/messages.po based on messages.pot
```

`pybabel init`命令将*messages.pot*文件作为输入，并将语言目录写入`-d`选项中指定的目录中，`-l`选项中指定的是翻译语言。 我将在*app/translations*目录中安装所有翻译，因为这是Flask-Babel默认提取翻译文件的地方。 该命令将在该目录内为西班牙数据文件创建一个*es*子目录。 特别是，将会有一个名为*app/translations/es/LC_MESSAGES/messages.po*的新文件，是需要翻译的文件路径。

如果你想支持其他语言，只需要各自的语言代码重复上述命令，就能使得每种语言都有一个包含*messages.po*文件的存储库。

在每个语言存储库中创建的`messages.po`文件使用的格式是语言翻译的事实标准，使用的格式为gettext。 以下是西班牙语*messages.po*开头的若干行：

```
# Spanish translations for PROJECT.
# Copyright (C) 2017 ORGANIZATION
# This file is distributed under the same license as the PROJECT project.
# FIRST AUTHOR <EMAIL@ADDRESS>, 2017.
#
msgid ""
msgstr ""
"Project-Id-Version: PROJECT VERSION\n"
"Report-Msgid-Bugs-To: EMAIL@ADDRESS\n"
"POT-Creation-Date: 2017-09-29 23:23-0700\n"
"PO-Revision-Date: 2017-09-29 23:25-0700\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language: es\n"
"Language-Team: es <LL@li.org>\n"
"Plural-Forms: nplurals=2; plural=(n != 1)\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=utf-8\n"
"Content-Transfer-Encoding: 8bit\n"
"Generated-By: Babel 2.5.1\n"

#: app/email.py:21
msgid "[Microblog] Reset Your Password"
msgstr ""

#: app/forms.py:12 app/forms.py:19 app/forms.py:50
msgid "Username"
msgstr ""

#: app/forms.py:13 app/forms.py:21 app/forms.py:43
msgid "Password"
msgstr ""
```

如果你跳过首段，可以看到后面的是从 `_()` 和 `_l()` 调用中提取的字符串列表。 对每个文本，都会展示其在应用中的引用位置。 然后， `msgid` 行包含原始语言的文本，后面的 `msgstr` 行包含一个空字符串。 这些空字符串需要被编辑，以使目标语言中的文本内容被填充。

有很多翻译应用程序与 `.po` 文件一起工作。 如果你擅长编辑文本文件，量少的时候也就罢了，但如果你正在处理大型项目，可能会推荐使用专门的编辑器。 最流行的翻译应用程序是开源的poedit，可用于所有主流操作系统。 如果你熟悉vim，那么po.vim 插件会提供一些键值映射，使得处理这些文件更加轻松。

在添加翻译后，你可以在下面看到一部分西班牙语*messages.po*：

```
#: app/email.py:21
msgid "[Microblog] Reset Your Password"
msgstr "[Microblog] Nueva Contraseña"

#: app/forms.py:12 app/forms.py:19 app/forms.py:50
msgid "Username"
msgstr "Nombre de usuario"

#: app/forms.py:13 app/forms.py:21 app/forms.py:43
msgid "Password"
msgstr "Contraseña"
```

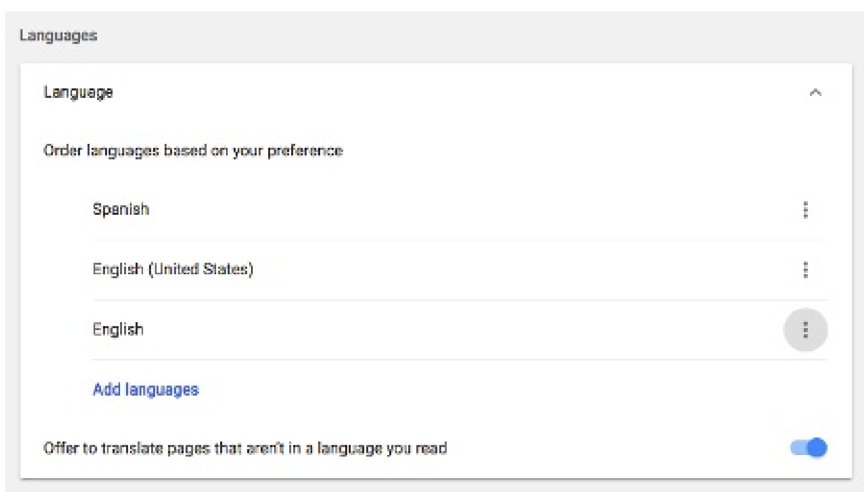
本章的下载包中包含所有翻译，此文件当然也在其中，所以你不必担心这部分的翻译工作。

*messages.po*文件是一种用于翻译的源文件。 当你想开始使用这些翻译后的文本时，这个文件需要被编译成一种格式，这种格式在运行时可以被应用程序使用。 要编译应用程序的所有翻译，可以使用 `pybabel compile` 命令，如下所示：

```
(venv) $ pybabel compile -d app/translations
compiling catalog app/translations/es/LC_MESSAGES/messages.po to
app/translations/es/LC_MESSAGES/messages.mo
```

此操作在每个语言存储库中的*messages.po*旁边添加*messages.mo*文件。 *.mo*文件是Flask-Babel将用于为应用程序加载翻译的文件。

在为西班牙语或任何其他添加到项目中的语言创建*messages.mo*文件之后，可以在应用中使用这些语言。 如果你想查看应用程序以西班牙语显示的方式，则可以在Web浏览器中编辑语言配置，以将西班牙语作为首选语言。 对Chrome，这是设置页面的高级部分：



如果你不想更改浏览器设置，另一种方法是通过使 `localeselector` 函数始终返回一种语言来强制实现。 对西班牙语，你可以这样做：

`app/__init__.py`：选择最佳语言。

```
@babel.localeselector
def get_locale():
    # return request.accept_languages.best_match(app.config['LANGUAGES'])
    return 'es'
```

使用为西班牙语配置的浏览器运行该应用或返回 `es` 的 `localeselector` 函数，将使所有文本在使用该应用时显示为西班牙文。

更新翻译

处理翻译时的一个常见情况是，即使翻译文件不完整，你也可能要开始使用翻译文件。 这是非常好的，你可以编译一个不完整的*messages.po*文件，任何可用的翻译都将被使用，而任何缺失的部分将使用原始语言。 随后，你可以继续处理翻译并再次编译，以便在取得进展时

更新*messages.mo*文件。

如果在添加 `_()` 包装器时错过了一些文本，则会出现另一种常见情况。 在这种情况下，你会发现你错过的那些文本将保持为英文，因为Flask-Babel对他们一无所知。 当你检测到这种情况时，会想要将其用 `_()` 或 `_l()` 包装，然后执行更新过程，这包括两个步骤：

```
(venv) $ pybabel extract -f babel.cfg -k _l -o messages.pot .
(venv) $ pybabel update -i messages.pot -d app/translations
```

`extract` 命令与我之前执行的命令相同，但现在它会生成*messages.pot*的新版本，其中包含所有以前的文本以及最近用 `_()` 或 `_l()` 包装的文本。 `update` 调用采用新的 `messages.pot` 文件并将其合并到与项目相关的所有*messages.po*文件中。 这将是一个智能合并，其中任何现有的文本将被单独保留，而只有在*messages.pot*中添加或删除的条目才会受到影响。

*messages.po*文件更新后，你就可以继续新的测试了，再次编译它，以便对应用生效。

翻译日期和时间

现在，我已经为Python代码和模板中的所有文本提供了完整的西班牙语翻译，但是如果你使用西班牙语运行应用并且是一个很好的观察者，那么会注意到还有一些内容以英文显示。 我指的是由Flask-Moment和moment.js生成的时间戳，显然这些时间戳并未包含在翻译工作中，因为这些包生成的文本都不是应用程序源代码或模板的一部分。

moment.js库确实支持本地化和国际化，所以我需要做的就是配置适当的语言。 Flask-Babel通过 `get_locale()` 函数返回给定请求的语言和语言环境，所以我要做的就是将语言环境添加到 `g` 对象，以便我可以从基础模板中访问它：

app/routes.py：存储选择的语言到flask.g中。

```
# ...
from flask import g
from flask_babel import get_locale

# ...

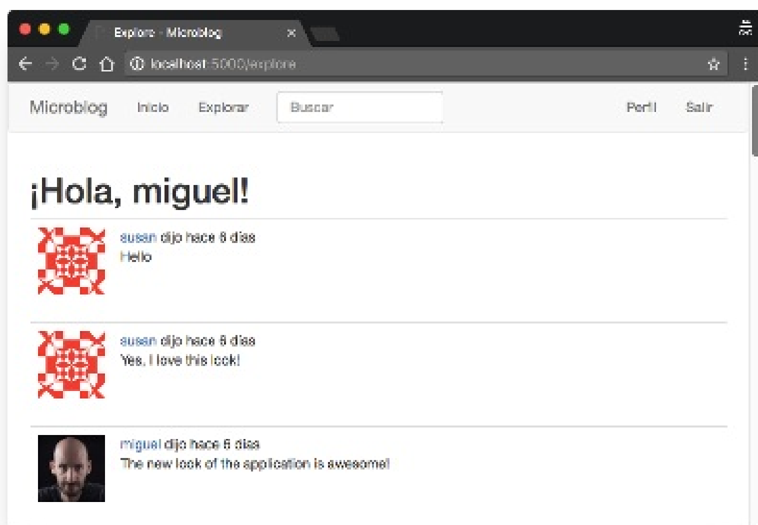
@app.before_request
def before_request():
    # ...
    g.locale = str(get_locale())
```

Flask-Babel的 `get_locale()` 函数返回一个本地语言对象，但我只想获得语言代码，可以通过将该对象转换为字符串来获取语言代码。 现在我有了 `g.locale` ，可以从基础模板中访问它，并以正确的语言配置moment.js：

app/templates/base.html：为moment.js设置本地语言

```
...
{% block scripts %}
    {{ super() }}
    {{ moment.include_moment() }}
    {{ moment.lang(g.locale) }}
{% endblock %}
```

现在所有的日期和时间都与文本使用相同的语言了。 你可以在下面看到西班牙语的外观：



此时，除用户在用户动态或个人资料说明中提供的文本外，所有其他的文本均可翻译成其他语言。

命令行增强

你可能会同意我的看法，pybabel命令有点长，难以记忆。 我将利用这个机会向你展示如何创建与 `flask` 命令集成的自定义命令。 到目前为止，你已经看到我使用Flask-Migrate扩展提供的 `flask run` 、 `flask shell` 和几个 `flask db` 子命令。 将应用特定的命令添加到 `flask` 实际上也很容易。 所以我现在要做的就是创建一些简单的命令，并用这个应用特有的参数触发 `pybabel` 命令。 我要添加的命令是：

- `flask translate init LANG` 用于添加新语言
- `flask translate update` 用于更新所有语言存储库
- `flask translate compile` 用于编译所有语言存储库

`babel export` 步骤不会设置为一个命令，因为生成*messages.pot*文件始终是运行 `init` 或 `update` 命令的先决条件，因此这些命令的执行将会生成翻译模板文件作为临时文件。

Flask依赖Click进行所有命令行操作。 像 `translate` 这样的命令是几个子命令的根，它们是通过 `app.cli.group()` 装饰器创建的。 我将把这些命令放在一个名为*app/cli.py*的新模块中：

app/cli.py：翻译命令组

```
from app import app

@app.cli.group()
def translate():
    """Translation and localization commands."""
    pass
```

该命令的名称来自被装饰函数的名称，并且帮助消息来自文档字符串。 由于这是一个父命令，它的存在只为子命令提供基础，函数本身不需要执行任何操作。

`update`和`compile`很容易实现，因为它们没有任何参数：

app/cli.py：更新子命令和编译子命令：

```
import os

# ...

@translate.command()
def update():
    """Update all languages."""
    if os.system('pybabel extract -F babel.cfg -k _l -o messages.pot .'):
        raise RuntimeError('extract command failed')
    if os.system('pybabel update -i messages.pot -d app/translations'):
        raise RuntimeError('update command failed')
    os.remove('messages.pot')

@translate.command()
def compile():
    """Compile all languages."""
    if os.system('pybabel compile -d app/translations'):
        raise RuntimeError('compile command failed')
```

请注意，这些函数的装饰器是如何从`translate`父函数派生的。 这似乎令人困惑，因为`translate()`是一个函数，但它是Click构建命令组的标准方式。 与`translate()`函数相同，这些函数的文档字符串在`--help`输出中用作帮助消息。

你可以看到，对于所有命令，运行它们并确保返回值为零（这意味着命令没有返回任何错误）。 如果命令错误，那么我会引发一个`RuntimeError`，这会导致脚本停止。 `update()`函数在同一个命令中结合了`extract`和`update`步骤，如果一切都成功的话，它会在更新完成后删除*messages.pot*文件，因为当再次需要这个文件时，可以很容易地重新生成 。

`init`命令将新的语言代码作为参数。 这是其执行流程：

app/cli.py：Init子命令。

```
import click

@translate.command()
@click.argument('lang')
def init(lang):
    """Initialize a new language."""
    if os.system('pybabel extract -F babel.cfg -k _l -o messages.pot .'):
        raise RuntimeError('extract command failed')
    if os.system(
            'pybabel init -i messages.pot -d app/translations -l ' + lang):
        raise RuntimeError('init command failed')
    os.remove('messages.pot')
```

该命令使用 `@click.argument` 装饰器来定义语言代码。 Click将命令中提供的值作为参数传递给处理函数，然后将该参数并入到 `init` 命令中。

启用这些命令的最后一步是导入它们，以便注册命令。 我决定在顶级目录的*microblog.py*文件中执行此操作：

microblog.py：注册命令。

```
from app import cli
```

这里我唯一需要做的就是导入新的*cli.py*模块，不需要做任何事情，因为导入操作会导致命令装饰器运行并注册命令。

此时，运行 `flask --help` 将列出 `translate` 命令作为选项。 `flask translate --help` 将显示我定义的三个子命令：

```
(venv) $ flask translate --help
Usage: flask translate [OPTIONS] COMMAND [ARGS]...

  Translation and localization commands.

Options:
  --help  Show this message and exit.

Commands:
  compile  Compile all languages.
  init     Initialize a new language.
  update   Update all languages.
```

所以现在工作流程就简便多了，而且不需要记住长而复杂的命令。 要添加新的语言，请使用：

```
(venv) $ flask translate init <language-code>
```

在更改 `_()` 和 `_l()` 语言标记后更新所有语言：

```
(venv) $ flask translate update
```

在更新翻译文件后编译所有语言：

```
(venv) $ flask translate compile
```

本文翻译自The Flask Mega-Tutorial Part XIV: Ajax

这是Flask Mega-Tutorial系列的第十四部分，我将使用Microsoft翻译服务和少许JavaScript来添加实时语言翻译功能。

在本章中，我将从服务器端开发的“安全区域”脱离，研究与服务器端同样重要的客户端组件的功能。 你是否看到过某些网站在用户生成的内容旁边显示的“翻译”链接？ 这些链接会触发非用户本地语言内容的实时自动翻译。 翻译的内容通常插入原始版本的下方。 Google将其显示为外语搜索结果。 Facebook在用户动态上使用它。 Twitter在推文上使用它。 今天我将向你展示如何将相同的功能添加到Microblog！

本章的*GitHub*链接为：*Browse*, *Zip*, *Diff.*

服务器端与客户端

迄今为止，在我遵循的传统服务器端模型中，有一个客户端（由用户驱动的Web浏览器）向应用服务器发出HTTP请求。 请求可以简单地请求HTML页面，例如当你单击“个人主页”链接时，或者它可以触发一个操作，例如在编辑你的个人信息之后单击提交按钮。 在这两种类型的请求中，服务器通过直接发送新的网页或通过发送重定向来完成请求。 然后客户端用新的页面替换当前页面。 只要用户停留在应用的网站上，该周期就会重复。 在这种模式下，服务器完成所有工作，而客户端只显示网页并接受用户输入。

有一种不同的模式，客户端扮演更积极的角色。 在这个模式中，客户端向服务器发出一个请求，服务器响应一个网页，但与前面的情况不同，并不是所有的页面数据都是HTML，页面中也有部分代码，通常用Javascript编写。 一旦客户端收到该页面，它就会显示HTML部分，并执行代码。 从那时起，你就拥有了一个可以独立工作的活动客户端，而无需与服务器进行联系或只有很少联系。 在严格的客户端应用中，整个应用通过初始页面请求下载到客户端，然后应用完全在客户端上运行，只有在查询或者变更数据时才与服务器联系。 这种类型的应用称为单页应用（SPAs）。

大多数应用是这两种模式的混合，并结合了两者的技术特点。 我的Microblog应用主要是服务器端应用，但今天我将添加一些客户端操作。 为了实时翻译用户动态，客户端浏览器将异步请求发送到服务器，服务器将响应该请求而不会导致页面刷新。然后客户端将动态地将翻译插入当前页面。 这种技术被称为Ajax)，这是Asynchronous JavaScript和XML的简称（尽管现在XML常常被JSON取代）。

实时翻译的工作流程

由于使用了Flask-Babel，本应用对外语有很好的支持，可以支持尽可能多的语言，只要我找到了对应的译文。 但是遗漏了一个元素，用户将会用他们自己的语言发表动态，所以用户很可能会用应用未知的语言发表动态。 自动翻译的质量大多数情况下不怎么样，但在，如果你只想对另一种语言的文本了解其基本含义，这已经足够了。

这正是Ajax大展身手的好机会！ 设想主页或发现页面可能会显示若干用户动态，其中一些可能是外语。 如果我使用传统的服务器端技术实现翻译，则翻译请求会导致原始页面被替换为新页面。 事实是，要求翻译诸多用户动态中的一条，并不是一个足够大的动作来要求整个页面的更新，如果翻译文本可以被动态地插入到原始文本下方，而剩下的页面保持原样，则用户体验更加出色。

实施实时自动翻译需要几个步骤。 首先，我需要一种方法来识别要翻译的文本的源语言。 我还需要知道每个用户的首选语言，因为我想仅为使用其他语言发表的动态显示“翻译”链接。 当提供翻译链接并且用户点击它时，我需要将Ajax请求发送到服务器，服务器将联系第三方翻译API。 一旦服务器发送了带有翻译文本的响应，客户端JavaScript代码将动态地将该文本插入到页面中。 你一定注意到了，这里有一些特殊的问题。 我将逐一审视这些问题。

语言识别

第一个问题是确定一条用户动态的语言。这不是一门精确的科学，因为不能确保监测结果绝对正确，但是对于大多数情况，自动检测的效果相当好。 在Python中，有一个称为`guess_language`的语言检测库，还算好用。 这个软件包的原始版本相当陈旧，从未被移植到Python 3，因此我将安装支持Python 2和3的派生版本：

```
(venv) $ pip install guess-language_spirit
```

计划是将每条用户动态提供给这个包，以尝试确定语言。 由于做这种分析有点费时，我不想每次把帖子呈现给页面时重复这项工作。 我要做的是在提交时为帖子设置源语言。 检测到的语言将被存储在post表中。

第一步，添加`language`字段到`Post`模型：

app/models.py：添加监测到的语言到`Post`模型：

```
class Post(db.Model):
    # ...
    language = db.Column(db.String(5))
```

你一定还记得，每当数据库模型发生变化时，都需要生成数据库迁移：

```
(venv) $ flask db migrate -m "add language to posts"
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.autogenerate.compare] Detected added column 'post.language'
  Generating migrations/versions/2b017edaa91f_add_language_to_posts.py ... done
```

然后将迁移应用到数据库：

```
(venv) $ flask db upgrade
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.runtime.migration] Upgrade ae346256b650 -> 2b017edaa91f, add language t
o posts
```

我现在可以在提交帖子时检测并存储语言：

app/routes.py：为新的用户动态保存语言字段。

```
from guess_language import guess_language

@app.route('/', methods=['GET', 'POST'])
@app.route('/index', methods=['GET', 'POST'])
@login_required
def index():
    form = PostForm()
    if form.validate_on_submit():
        language = guess_language(form.post.data)
        if language == 'UNKNOWN' or len(language) > 5:
            language = ''
        post = Post(body=form.post.data, author=current_user,
                    language=language)
        # ...
```

有了这个变更，每次发表动态时，都会通过 `guess_language` 函数测试文本来尝试确定语言。如果语言监测为未知，或者如果我得到意想不到的长字符串的结果，我会将一个空字符串保存到数据库中以安全地使用它。 我将采用约定，将任何将把语言设置为空字符串的帖子假定为未知语言。

展示一个“翻译”链接

第二步很简单。 我现在要做的是在任何不是当前用户的首选语言的用户动态下，添加一个“翻译”链接。

app/templates/_post.html：给用户动态添加翻译链接。

```
                {% if post.language and post.language != g.locale %}
                <br><br>
                <a href="#">{{ _('Translate') }}</a>
                {% endif %}
```

我在 `_post.html` 子模板中执行此操作，以便此功能出现在显示用户动态的任何页面上。 翻译链接只会出现在检测到语言种类的动态下，并且必须满足的条件是，这种语言与用Flask-Babel的 `localeselector` 装饰器装饰的函数选择的语言不匹配。 回想一下第十三章所选语言环境存储为 `g.locale` 。 链接文本需要以Flask-Babel可以翻译的方式添加，所以我在定义它时使用了 `_()` 函数。

请注意，我还没有关联此链接的操作。 首先，我想弄清楚如何进行实际的翻译。

使用第三方翻译服务

两种主要的翻译服务是Google Cloud Translation API和Microsoft Translator Text API。 两者都是付费服务，但微软为低频少量的翻译提供了免费的入门级选项。 谷歌过去提供免费翻译服务，但现在，即使是最低层次的服务也需要付费。 因为我希望能够在不产生费用的情况下尝试翻译，我将实施Microsoft的解决方案。

在使用Microsoft Translator API之前，你需要先获得微软云服务Azure的帐户。 你可以选择免费套餐，但在注册过程中系统会要求你提供信用卡号，但在你保持该级别的服务时，你的卡不会被收取费用。

获得Azure帐户后，转到Azure门户并单击左上角的“New”按钮，然后键入或选择“Translator Text API”。 当你点击“Create”按钮时，将看到一个表单，并可以在其中定义一个新的翻译器资源，然后将其添加到你的帐户中。 你可以在下面看到我是如何完成表单的：

Create
Translator Text API
* Name
translator
* Subscription
Free Trial
* Pricing tier (View full pricing details)
F0 (2M Characters translated per month)
* Resource group
Create new Use existing
translator
* Resource group location
Central US
* I confirm I have read and understood the notice below.

当你再次点击“Create”按钮时，翻译器API资源将被添加到你的帐户中。几秒钟之后，你将在顶栏中收到通知，说明部署了翻译器资源。 点击通知中的“Go to resource”按钮，然后点击左侧栏上的“Keys”选项。 你现在将看到两个Key，分别标记为“Key 1”和“Key 2”。 将其中一个Key复制到剪贴板，然后将其设置到终端的环境变量中（如果使用的是Microsoft Windows，请用 `set` 替换 `export` ）：

```
(venv) $ export MS_TRANSLATOR_KEY=<paste-your-key-here>
```

该Key用于验证翻译服务，因此需要将其添加到应用配置中：

config.py: 添加Microsoft Translator API key到配置中。

```
class Config(object):
    # ...
    MS_TRANSLATOR_KEY = os.environ.get('MS_TRANSLATOR_KEY')
```

与很多配置值一样，我更喜欢将它们安装在环境变量中，并从那里将它们导入到Flask配置中。 对于允许访问第三方服务的密钥或密码等敏感信息，这一点尤为重要。 你绝对不想在代码中明确写出它们。

Microsoft Translator API是一个接受HTTP请求的Web服务。 Python中有若干HTTP客户端，但最常用和最简单的就是 `requests` 包。 所以让我们将其安装到虚拟环境中：

```
(venv) $ pip install requests
```

在下面，你可以看到我使用Microsoft Translator API编写翻译文本的功能。 我来新增一个*app/translate.py*模块：

app/translate.py：文本翻译函数。

```
import json
import requests
from flask_babel import _
from app import app

def translate(text, source_language, dest_language):
    if 'MS_TRANSLATOR_KEY' not in app.config or \
            not app.config['MS_TRANSLATOR_KEY']:
        return _('Error: the translation service is not configured.')
    auth = {'Ocp-Apim-Subscription-Key': app.config['MS_TRANSLATOR_KEY']}
    r = requests.get('https://api.microsofttranslator.com/v2/Ajax.svc'
                     '/Translate?text={}&from={}&to={}'.format(
                         text, source_language, dest_language),
                     headers=auth)
    if r.status_code != 200:
        return _('Error: the translation service failed.')
    return json.loads(r.content.decode('utf-8-sig'))
```

该函数定义需要翻译的文本、源语言和目标语言为参数，并返回翻译后文本的字符串。 它首先检查配置中是否存在翻译服务的Key，如果不存在，则会返回错误。 错误也是一个字符串，所以从外部看，这将看起来像翻译文本。 这可确保在出现错误时用户将看到有意义的错误消息。

`requests` 包中的 `get()` 方法向作为第一个参数给定的URL发送一个带有GET方法的HTTP请求。 我使用*/v2/Ajax.svc/Translate* URL，它是翻译服务中的一个端点，它将翻译内容荷载为JSON返回。文本、源语言和目标语言都需要在URL中分别命名为 `text` ， `from` 和 `to` 作为查询字符串参数。 要使用该服务进行身份验证，我需要将我添加到配置中的Key传递给该服务。 该Key需要在名为 `Ocp-Apim-Subscription-Key` 的自定义HTTP头中给出。 我创建了 `auth` 字典，然后将它通过 `headers` 参数传递给 `requests` 。

`requests.get()` 方法返回一个响应对象，它包含了服务提供的所有细节。 我首先需要检查和确认状态码是200，这是成功请求的代码。 如果我得到任何其他代码，我就知道发生了错误，所以在这种情况下，我返回一个错误字符串。 如果状态码是200，那么响应的主体就有一个带有翻译的JSON编码字符串，所以我需要做的就是使用Python标准库中的 `json.loads()` 函数将JSON解码为我可以使用的Python字符串。 响应对象的 `content` 属性包含作为字节对象的响应的原始主体，该属性是UTF-8编码的字符序列，需要先进行解码，然后发送给 `json.loads()` 。

下面你可以看到一个Python控制台会话，我演示了如何使用新的 `translate()` 函数：

```
>>> from app.translate import translate
>>> translate('Hi, how are you today?', 'en', 'es')  # English to Spanish
'Hola, ¿cómo estás hoy?'
>>> translate('Hi, how are you today?', 'en', 'de')  # English to German
'Are Hallo, how you heute?'
>>> translate('Hi, how are you today?', 'en', 'it')  # English to Italian
'Ciao, come stai oggi?'
>>> translate('Hi, how are you today?', 'en', 'fr')  # English to French
"Salut, comment allez-vous aujourd'hui ?"
```

很酷，对吧？ 现在是时候将此功能与应用集成在一起了。

来自服务器的Ajax

我将从实现服务器端部分开始。 当用户单击动态下方显示的翻译链接时，将向服务器发出异步HTTP请求。 我将在下一节中向你展示如何执行此操作，因此现在我将专注于实现服务器处理此请求的操作。

异步（Ajax）请求类似于我在应用中创建的路由和视图函数，唯一的区别是它不返回HTML或重定向，而是返回数据，格式为XML或更常见的JSON。 你可以在下面看到翻译视图函数，该函数调用Microsoft Translator API，然后返回JSON格式的翻译文本：

app/routes.py：文本翻译视图函数。

```
from flask import jsonify
from app.translate import translate

@app.route('/translate', methods=['POST'])
@login_required
def translate_text():
    return jsonify({'text': translate(request.form['text'],
                                      request.form['source_language'],
                                      request.form['dest_language'])})
```

如你所见，相当简单。 我以 `POST` 请求的形式实现了这条路由。 关于什么时候使用 `GET` 或 `POST` （或者还没有见过的其他请求方法），真的没有绝对的规则。 由于客户端将发送数据，因此我决定使用 `POST` 请求，因为它与提交表单数据的请求类似。

`request.form`属性是Flask用提交中包含的所有数据暴露的字典。 当我使用Web表单工作时，我不需要查看`request.form`，因为Flask-WTF可以为我工作，但在这种情况下，实际上没有Web表单，所以我必须直接访问数据。

所以我在这个函数中做的是调用上一节中的`translate()`函数，直接从通过请求提交的数据中传递三个参数。 将结果合并到单个键`text`下的字典中，字典作为参数传递给Flask的`jsonify()`函数，该函数将字典转换为JSON格式的有效载荷。 `jsonify()`返回的值是将被发送回客户端的HTTP响应。

例如，如果客户希望将字符串“Hello，World！”翻译成西班牙语，则来自该请求的响应将具有以下有效载荷：

```
{ "text": "Hola, Mundo!" }
```

来自客户端的**Ajax**

因此，现在服务器能够通过*/translate* URL提供翻译，当用户单击我上面添加的“翻译”链接时，我需要调用此URL，传递需要翻译的文本、源语言和目标语言。 如果你不熟悉在浏览器中使用JavaScript，这将是一个很好的学习机会。

在浏览器中使用JavaScript时，当前显示的页面在内部被表示为文档对象模型（DOM）。 这是一个引用页面中所有元素的层次结构。 在此上下文中运行的JavaScript代码可以更改DOM以触发页面中的更改。

我们首先需要讨论的是，在浏览器中运行的JavaScript代码如何获取需要发送到服务器中运行的翻译函数的三个参数。 为了获得文本，我需要找到包含用户动态正文的DOM内的节点并获取它的内容。 为了便于识别包含用户动态的DOM节点，我将为它们附加一个唯一的ID。 如果你查看*_post.html*模板，则呈现用户动态正文的行只会读取`{{post.body}}`。 我要做的是将这些内容包装在一个`<span>`元素中。 这不会在视觉上改变任何东西，但它给了我一个可以插入标识符的地方：

app/templates/_post.html：给每条用户动态添加ID。

```
<span id="post{{ post.id }}">{{ post.body }}</span>
```

这将为每条用户动态分配一个唯一标识符，格式为`post1`，`post2`等，其中数字与每条用户动态的数据库标识符相匹配。 现在每条用户动态都有一个唯一的标识符，给定一个ID值，我可以使用jQuery定位`<span>`元素并提取其中的文本。 例如，如果我想获得ID为123的用户动态的文本，我可以这样做：

```
$('#post123').text()
```

这里的$符号是jQuery库提供的函数的名称。 这个库被Bootstrap使用，所以它已经被Flask-Bootstrap包含。 `#` 是jQuery使用的“选择器”语法的一部分，这意味着接下来是元素的ID。

我也希望有一个地方可以在我从服务器收到翻译文本后插入翻译文本。 我要做的是将“翻译”链接替换为翻译文本，因此我还需要为该节点提供唯一标识符：

app/templates/_post.html：为翻译链接添加ID。

```
                <span id="translation{{ post.id }}">
                    <a href="#">{{ _('Translate') }}</a>
                </span>
```

因此，现在对于一个给定的用户动态ID，我有一个用于用户动态的 `post <ID>` 节点和一个对应的 `translation <ID>` 节点，我可以在用翻译后的文本替换翻译链接时用到它们。

下一步是编写一个可以完成所有翻译工作的函数。 该函数将利用输入和输出DOM节点以及源语言和目标语言，向服务器发出携带必须的三个参数的异步请求，并在服务器响应后用翻译后的文本替换翻译链接。 这听起来像很多工作，但实现相当简单：

app/templates/base.html：客户端翻译函数。

```
{% block scripts %}
    ...
    <script>
        function translate(sourceElem, destElem, sourceLang, destLang) {
            $(destElem).html('<img src="{{ url_for('static', filename='loading.gif') }
}">');
            $.post('/translate', {
                text: $(sourceElem).text(),
                source_language: sourceLang,
                dest_language: destLang
            }).done(function(response) {
                $(destElem).text(response['text'])
            }).fail(function() {
                $(destElem).text("{{ _('Error: Could not contact server.') }}");
            });
        }
    </script>
{% endblock %}
```

前两个参数是用户动态和翻译链接节点的唯一ID，后两个参数是源语言和目标语言代码。

该函数从一个很好的接触开始：它添加一个加载器替换翻译链接，以便用户知道翻译正在进行中。 这是通过使用 `$(destElem).html()` 函数完成的，它用基于 `<img>` 元素的新HTML内容替换定义为翻译链接的原始HTML。 对于加载器，我将使用一个小的动画GIF，它已添加到Flask为静态文件保留的*app/static*目录中。 为了生成引用这个图像的URL，我使用 `url_for()` 函数，传递特殊的路由名称 `static` 并给出图像的文件名作为参数。 你可以在本章的下载包中找到*loading.gif*图像。

现在我用一个优雅的加载器代替了翻译链接，以便用户知道要等待翻译出现。 下一步是将POST请求发送到我在前一节中定义的*/translate* URL。 为此，我也将使用jQuery，本处使用 `$ .post()` 函数。 这个函数以一种类似于浏览器提交Web表单的格式向服务器提交数据，

这很方便，因为它允许Flask将这些数据合并到 `request.form` 字典中。 `$ .post()` 的参数是两个，第一个是发送请求的URL，第二个是包含服务器期望的三个数据项的字典（或者称之为对象，因为这些是在JavaScript中调用的）。

你可能知道JavaScript对回调函数（或者称为*promises*的更高级的回调形式）友好。 现在要做的就是说明一旦这个请求完成并且浏览器接收到响应，我想完成的事情。 在JavaScript中没有需要等待的事情，一切都是异步。 我需要做的是提供一个回调函数，浏览器在接收到响应时调用它。 而且，为了使所有内容尽可能健壮，我想指出在出现错误的情况下该怎么做，以作为处理错误的第二个回调函数。 有几种方法可以指定这些回调，但在这种情况下，使用promises可以使代码更加清晰。 语法如下：

```
$.post(<url>, <data>).done(function(response) {
    // success callback
}).fail(function() {
    // error callback
})
```

promise语法允许将 `$ .post()` 调用的返回值“传入”回调函数作为参数。 在成功回调中，我所需要做的就是使用翻译后的文本调用 `$(destElem).text()` ，该文本在字典中 `text` 键下。 在出现错误的情况下，我也是这样做的，但是我显示的文本是一条通用的错误消息，我会确保它会作为可翻译的文本编入基础模板中。

所以现在唯一剩下的就是通过用户点击翻译链接来触发具有正确参数的 `translate()` 函数。 存在若干方法可以做到这一点，我要做的是将该函数的调用嵌入链接的 `href` 属性中：

app/templates/_post.html：翻译链接处理器。

```
                <span id="translation{{ post.id }}">
                    <a href="javascript:translate(
                                '#post{{ post.id }}',
                                '#translation{{ post.id }}',
                                '{{ post.language }}',
                                '{{ g.locale }}');">{{ _('Translate') }}</a>
                </span>
```

链接的 `href` 元素可以接受任何JavaScript代码，如果它带有 `javascript:` 前缀的话，那么这是一种方便的方式来调用翻译函数。 因为这个链接将在客户端请求页面时在服务器端渲染，所以我可以使用 `{{}}` 表达式来为函数生成四个参数。 每条用户动态都有自己的翻译链接，以及其唯一生成的参数。 `post <ID>` 和 `translation <ID>` 需要渲染具体的ID，它们都需要在被使用时加上 `#` 前缀。

现在实时翻译功能已经完成！ 如果你在环境中设置了有效的Microsoft Translator API Key，则现在应该能够触发翻译。 假设你的浏览器设置为偏好英语，则需要使用其他语言撰写文章以查看“翻译”链接。 下面你可以看到一个例子：



在本章中，我介绍了一些需要翻译成应用支持的所有语言的新文本，因此有必要更新翻译目录：

```
(venv) $ flask translate update
```

对于你自己的项目，需要编辑每个语言存储库中的*messages.po*文件以包含这些新测试的翻译，不过我已经在本章的下载包或GitHub存储库中创建了西班牙语翻译。

要完成新的翻译，还需要执行编译：

```
(venv) $ flask translate compile
```

本文翻译自The Flask Mega-Tutorial Part XV: A Better Application Structure

这是Flask Mega-Tutorial系列的第十五部分，我将使用适用于大型应用的风格重构本应用。

Microblog已经是一个初具规模的应用了，所以我认为这是讨论Flask应用如何在持续增长中不会变得混乱和难以管理的好时机。 Flask是一个框架，旨在让你选择以任何方式来组织项目，基于该理念，在应用日益庞大或者技能水平变化的时候，才有可能更改和调整其结构。

在本章中，我将讨论适用于大型应用的一些模式，并且为了演示他们，我将对Microblog项目的结构进行一些更改，目标是使代码更易于维护和组织。 当然，在真正的Flask精神中，我鼓励你在尝试决定组织自己的项目的方式时仅仅将这些更改作为参考。

本章的*GitHub*链接为：*Browse*, *Zip*, *Diff.*

目前的局限性

目前状态下的应用有两个基本问题。 如果你观察应用的组织方式，你会注意到有几个不同的子系统可以被识别，但支持它们的代码都混合在了一起，没有任何明确的界限。 我们来回顾一下这些子系统是什么：

- 用户认证子系统，包括*app/routes.py*中的一些视图函数，*app/forms.py*中的一些表单，*app/templates*中的一些模板以及*app/email.py中的电子邮件支持。
- 错误子系统，它在*app/errors.py*中定义了错误处理程序并在*app/templates*中定义了模板。
- 核心应用功能，包括显示和撰写用户动态，用户个人主页和关注以及用户动态的实时翻译，这些功能遍布大多数应用模块和模板。

思考这三个子系统以及它们组织的方式，你可能会注意到这样一个模式。 到目前为止，我一直遵循的组织逻辑是不同的应用功能归属到其专属的模块。 这些模块之中，一个用于视图函数，一个用于Web表单，一个用于错误，一个用于电子邮件，一个目录用于存放HTML模板等等。 虽然这是一个对小项目有意义的组织结构，但是一旦项目开始增长，它往往会使其中的一些模块变得非常大而且杂乱无章。

要想清晰地看到问题的一种方法，是思考如何通过尽可能多地重复使用这一项目来开始第二个项目。 例如，用户身份验证部分应该在其他应用中也能运行良好，但如果你想按原样使用该代码，则必须进入多个模块并将相关部分复制/粘贴到新项目的新文件中。 看到这是多么不方便了吗？ 如果这个项目将所有与认证相关的文件从应用的其余部分中分离出来，会不会更好？ Flask的*blueprints*功能有助于实现更实用的组织结构，从而更轻松地重用代码。

还有第二个问题，虽然它不太明显。 Flask应用实例在 `app/__init__.py` 中被创建为一个全局变量，然后又被很多应用模块导入。 虽然这本身并不是问题，但将应用实例作为全局变量可能会使某些情况复杂化，特别是与测试相关的情景。 想象一下你想要在不同的配置下测试这个应用。 由于应用被定义为全局变量，实际上没有办法使用不同配置变量来实例化的两个应

用实例。 另一种糟心的情况是，所有测试都使用相同的应用，因此测试可能会对应用进行更改，就会影响稍后运行的其他测试。 理想情况下，你希望所有测试都在原始应用实例上运行的。

你可以在*tests.py*模块中看到我正在使用的应用实例化之后修改配置的技巧，以指示测试时使用内存数据库而不是默认的SQLite数据库。我真的没有其他办法来更改已配置的数据库，因为在测试开始时已经创建和配置了应用。 对于这种特殊情况，对已配置的应用实例修改配置似乎可以运行，但在其他情况下可能不会，并且在任何情况下，这是一种不推荐的做法，因为这么做可能会导致提示晦涩并且难以找到BUG。

更好的解决方案是不将应用设置为全局变量，而是使用应用工厂函数在运行时创建它。 这将是一个接受配置对象作为参数的函数，并返回一个配置完毕的Flask应用实例。 如果我能够通过应用工厂函数来修改应用，那么编写需要特殊配置的测试会变得很容易，因为每个测试都可以创建它各自的应用。

在本章中，我将通过为上面提到的三个子系统重构应用来介绍blueprints。 展示更改的详细列表有些不切实际，因为几乎应用中每个文件都有少许变化，所以我将讨论重构的步骤，然后你可以以下载更改后的应用。

Blueprints

在Flask中，blueprint是代表应用子集的逻辑结构。 blueprint可以包括路由，视图函数，表单，模板和静态文件等元素。 如果在单独的Python包中编写blueprint，那么你将拥有一个封装了应用特定功能的组件。

Blueprint的内容最初处于休眠状态。 为了关联这些元素，blueprint需要在应用中注册。 在注册过程中，需要将添加到blueprint中的所有元素传递给应用。 因此，你可以将blueprint视为应用功能的临时存储，以帮助组织代码。

错误处理Blueprint

我创建的第一个blueprint用于封装对错误处理程序的支持。 该blueprint的结构如下：

```
app/
    errors/                             <-- blueprint package
        __init__.py                     <-- blueprint creation
        handlers.py                     <-- error handlers
    templates/
        errors/                         <-- error templates
            404.html
            500.html
    __init__.py                         <-- blueprint registration
```

实质上，我所做的是将*app/errors.py*模块移动到*app/errors/handlers.py*中，并将两个错误模板移动到*app/templates/errors*中，以便将它们与其他模板分开。 我还必须在两个错误处理程序中更改 `render_template()` 调用以使用新的*errors*模板子目录。 之后，我将blueprint创建添加到*app/errors/**init**.py*模块，并在创建应用实例之后，将blueprint注册到*app/**init**.py*。

我必须提一下，Flask blueprints可以为自己的模板和静态文件配置单独的目录。 我已决定将模板移动到应用模板目录的子目录中，以便所有模板都位于一个层次结构中，但是如果你希望在blueprint中包含属于自己的模板，这也是支持的。 例如，如果向 `Blueprint()` 构造函数添加 `template_folder='templates'` 参数，则可以将错误blueprint的模板存储在*app/errors/templates*目录中。

创建blueprint与创建应用非常相似。 这是在blueprint的 `__init__.py` 模块中完成的：

`app/errors/__init__.py` ：错误blueprint。

```
from flask import Blueprint

bp = Blueprint('errors', __name__)

from app.errors import handlers
```

`Blueprint` 类获取blueprint的名称，基础模块的名称（通常在Flask应用实例中设置为 `__name__` ）以及一些可选参数（在这种情况下我不需要这些参数）。 Blueprint对象创建后，我导入了*handlers.py*模块，以便其中的错误处理程序在blueprint中注册。 该导入位于底部以避免循环依赖。

在*handlers.py*模块中，我放弃使用 `@app.errorhandler` 装饰器将错误处理程序附加到应用程序，而是使用blueprint的 `@bp.app_errorhandler` 装饰器。 尽管两个装饰器最终都达到了相同的结果，但这样做的目的是试图使blueprint独立于应用，使其更具可移植性。我还需要修改两个错误模板的路径，因为它们被移动到了新*errors*子目录。

完成错误处理程序重构的最后一步是向应用注册blueprint：

*app/**init**.py*：向应用注册错误blueprint。

```
app = Flask(__name__)

# ...

from app.errors import bp as errors_bp
app.register_blueprint(errors_bp)

# ...

from app import routes, models  # <-- remove errors from this import!
```

为了注册blueprint，将使用Flask应用实例的 `register_blueprint()` 方法。 在注册blueprint时，任何视图函数，模板，静态文件，错误处理程序等均连接到应用。 我将blueprint的导入放在 `app.register_blueprint()` 的上方，以避免循环依赖。

用户认证Blueprint

将应用的认证功能重构为blueprint的过程与错误处理程序的过程非常相似。 以下是重构为blueprint的目录层次结构：

```
app/
    auth/                               <-- blueprint package
        __init__.py                     <-- blueprint creation
        email.py                        <-- authentication emails
        forms.py                        <-- authentication forms
        routes.py                       <-- authentication routes
    templates/
        auth/                           <-- blueprint templates
            login.html
            register.html
            reset_password_request.html
            reset_password.html
    __init__.py                         <-- blueprint registration
```

为了创建这个blueprint，我必须将所有认证相关的功能移到为blueprint创建的新模块中。 这包括一些视图函数，Web表单和支持功能，例如通过电子邮件发送密码重设token的功能。 我还将模板移动到一个子目录中，以将它们与应用的其余部分分开，就像我对错误页面所做的那样。

在blueprint中定义路由时，使用 `@bp.route` 装饰器来代替 `@app.route` 装饰器。 在 `url_for()` 中用于构建URL的语法也需要进行更改。 对于直接附加到应用的常规视图函数， `url_for()` 的第一个参数是视图函数名称。 但当在blueprint中定义路由时，该参数必须包含blueprint名称和视图函数名称，并以句点分隔。 因此，我不得不用诸如 `url_for('auth.login')` 的代码替换所有出现的 `url_for('login')` 代码，对于其余的视图函数也是如此。

注册 `auth` blueprint到应用时，我使用了些许不同的格式：

`app/__init__.py` ：注册用户认证blueprint到应用。

```
# ...
from app.auth import bp as auth_bp
app.register_blueprint(auth_bp, url_prefix='/auth')
# ...
```

在这种情况下， `register_blueprint()` 调用接收了一个额外的参数， `url_prefix` 。 这完全是可选的，Flask提供了给blueprint的路由添加URL前缀的选项，因此blueprint中定义的任何路由都会在其完整URL中获取此前缀。 在许多情况下，这可以用来当成“命名空间”，它可以将blueprint中的所有路由与应用或其他blueprint中的其他路由分开。 对于用户认证，我认为让所有路由以*/auth*开头很不错，所以我添加了该前缀。 所以现在登录URL将会是*http://localhost:5000/auth/login*。 因为我使用 `url_for()` 来生成URL，所有URL都会自动合并前缀。

主应用Blueprint

第三个blueprint包含核心应用逻辑。 重构这个blueprint和前两个blueprint的过程一样。 我给这个blueprint命名为 `main` ，因此所有引用视图函数的 `url_for()` 调用都必须添加一个 `main.` 前缀。 鉴于这是应用的核心功能，我决定将模板留在原来的位置。 这不会有什么问题，因为我已将其他两个blueprint中的模板移动到子目录中了。

应用工厂模式

正如我在本章的介绍中所提到的，将应用设置为全局变量会引入一些复杂性，主要是以某些测试场景的局限性为形式。 在我介绍blueprint之前，应用必须是一个全局变量，因为所有的视图函数和错误处理程序都需要使用来自 `app` 的装饰器来修饰，比如 `@app.route` 。 但是现在所有的路由和错误处理程序都被转移到了blueprint中，因此保持应用全局性的理由就不够充分了。

所以我要做的是添加一个名为 `create_app()` 的函数来构造一个Flask应用实例，并消除全局变量。 转换并非容易，我不得不理清一些复杂的东西，但我们先来看看应用工厂函数：

*app/**init**.py*：应用工厂函数。

```
# ...
db = SQLAlchemy()
migrate = Migrate()
login = LoginManager()
login.login_view = 'auth.login'
login.login_message = _l('Please log in to access this page.')
mail = Mail()
bootstrap = Bootstrap()
moment = Moment()
babel = Babel()

def create_app(config_class=Config):
    app = Flask(__name__)
    app.config.from_object(config_class)

    db.init_app(app)
    migrate.init_app(app, db)
    login.init_app(app)
    mail.init_app(app)
    bootstrap.init_app(app)
    moment.init_app(app)
    babel.init_app(app)

    # ... no changes to blueprint registration

    if not app.debug and not app.testing:
        # ... no changes to logging setup

    return app
```

你已经看到，大多数Flask插件都是通过创建插件实例并将应用作为参数传递来初始化的。 当应用不再作为全局变量时，有一种替代模式，插件分成两个阶段进行初始化。 插件实例首先像前面一样在全局范围内创建，但没有参数传递给它。 这会创建一个未附加到应用的插件实

例。 当应用实例在工厂函数中创建时，必须在插件实例上调用`init_app()`方法，以将其绑定到现在已知的应用。

在初始化期间执行的其他任务保持不变，但会被移到工厂函数而不是在全局范围内。 这包括blueprint和日志配置的注册。 请注意，我在条件中添加了一个`not app.testing`子句，用于决定是否启用电子邮件和文件日志，以便在单元测试期间跳过所有这些日志记录。 由于在配置中`TESTING`变量在单元测试时会被设置为`True`，因此`app.testing`标志在运行单元测试时将变为`True`。

那么谁来调用应用程工厂函数呢？ 最明显使用此函数的地方是处于顶级目录的*microblog.py*脚本，它是唯一会将应用设置为全局变量的模块。 另一个调用该工厂函数的地方是*tests.py*，我将在下一节中更详细地讨论单元测试。

正如我上面提到的，大多数对`app`的引用都是随着blueprint的引入而消失的，但是我仍然需要解决代码中的一些问题。 例如，*app/models.py*、*app/translate.py*和*app/main/routes.py*模块都引用了`app.config`。 幸运的是，Flask开发人员试图使视图函数很容易地访问应用实例，而不必像我一直在做的那样导入它。 Flask提供的`current_app`变量是一个特殊的“上下文”变量，Flask在分派请求之前使用应用初始化该变量。 你之前已经看到另一个上下文变量，即存储当前语言环境的`g`变量。 这两个变量，以及Flask-Login的`current_user`和其他一些你还没有看到的东西，是“魔法”变量，因为它们像全局变量一样工作，但只能在处理请求期间且在处理它的线程中访问。

用Flask的`current_app`变量替换`app`就不需要将应用实例作为全局变量导入。 通过简单的搜索和替换，我可以毫无困难地用`current_app.config`替换对`app.config`的所有引用。

*app/email.py*模块提出了一个更大的挑战，所以我必须使用一个小技巧：

app/email.py：将应用实例传递给另一个线程。

```
from app import current_app

def send_async_email(app, msg):
    with app.app_context():
        mail.send(msg)

def send_email(subject, sender, recipients, text_body, html_body):
    msg = Message(subject, sender=sender, recipients=recipients)
    msg.body = text_body
    msg.html = html_body
    Thread(target=send_async_email,
           args=(current_app._get_current_object(), msg)).start()
```

在`send_email()`函数中，应用实例作为参数传递给后台线程，后台线程将发送电子邮件而不阻塞主应用程序。在作为后台线程运行的`send_async_email()`函数中直接使用`current_app`将不会奏效，因为`current_app`是一个与处理客户端请求的线程绑定的上下文感知变量。在另一个线程中，`current_app`没有赋值。直接将`current_app`作为参数传递给线程对象也不会有效，因为`current_app`实际上是一个代理对象，它被动态地映射到应用实例。因此，传递代理

对象与直接在线程中使用`current_app`相同。我需要做的是访问存储在代理对象中的实际应用程序实例，并将其作为`app`参数传递。`current_app._get_current_object()`表达式从代理对象中提取实际的应用实例，所以它就是我作为参数传递给线程的。

另一个棘手的模块是*app/cli.py*，它实现了一些用于管理语言翻译的快捷命令。在这种情况下，`current_app`变量不起作用，因为这些命令是在启动时注册的，而不是在处理请求期间（这是唯一可以使用`current_app`的时间段）注册的。为了在这个模块中删除对`app`的引用，我使用了另一个技巧，将这些自定义命令移动到一个将`app`实例作为参数的`register()`函数中：

app/cli.py：注册自定义应用命令。

```
import os
import click

def register(app):
    @app.cli.group()
    def translate():
        """Translation and localization commands."""
        pass

    @translate.command()
    @click.argument('lang')
    def init(lang):
        """Initialize a new language."""
        # ...

    @translate.command()
    def update():
        """Update all languages."""
        # ...

    @translate.command()
    def compile():
        """Compile all languages."""
        # ...
```

然后我从*microblog.py*中调用这个`register()`函数。以下是完成重构后的*microblog.py*：

microblog.py：重构后的主应用模块。

```
from app import create_app, db, cli
from app.models import User, Post

app = create_app()
cli.register(app)

@app.shell_context_processor
def make_shell_context():
    return {'db': db, 'User': User, 'Post' :Post}
```

单元测试的改进

正如我在本章开头所暗示的，到目前为止，我所做的很多工作都是为了改进单元测试工作流程。在运行单元测试时，要确保应用的配置方式不会污染开发资源（如数据库）。

*tests.py*的当前版本采用了应用实例化之后修改配置的技巧，这是一种危险的做法，因为并不是所有类型的更改都会在修改之后才生效。我想要的是有机会在添加到应用之前指定我想要的测试配置项。

`create_app()` 函数现在接受一个配置类作为参数。默认情况下，使用在*config.py*中定义的 `Config` 类，但现在我可以通过将新类传递给工厂函数来创建使用不同配置的应用实例。下面是一个适用于我的单元测试的示例配置类：

tests.py：测试配置。

```
from config import Config

class TestConfig(Config):
    TESTING = True
    SQLALCHEMY_DATABASE_URI = 'sqlite://'
```

我在这里做的是创建应用的 `Config` 类的子类，并覆盖SQLAlchemy配置以使用内存SQLite数据库。我还添加了一个 `TESTING` 属性，并设置为 `True` ，我目前不需要该属性，但如果应用需要确定它是否在单元测试下运行，它就派上用场了。

你一定还记得，我的单元测试依赖于 `setUp()` 和 `tearDown()` 方法，它们由单元测试框架自动调用，以创建和销毁每次测试运行的环境。我现在可以使用这两种方法为每个测试创建和销毁一个测试专用的应用：

tests.py：为每次测试创建一个应用。

```
class UserModelCase(unittest.TestCase):
    def setUp(self):
        self.app = create_app(TestConfig)
        self.app_context = self.app.app_context()
        self.app_context.push()
        db.create_all()

    def tearDown(self):
        db.session.remove()
        db.drop_all()
        self.app_context.pop()
```

新的应用将存储在 `self.app` 中，但光是创建一个应用不足以使所有的工作都成功。思考创建数据库表的 `db.create_all()` 语句。 `db` 实例需要注册到应用实例，因为它需要从 `app.config` 获取数据库URI，但是当你使用应用工厂时，应用就不止一个了。那么 `db` 如何关联到我刚刚创建的 `self.app` 实例呢？

答案在*application context*中。还记得 `current_app` 变量吗？当不存在全局应用实例导入时，该变量以代理的形式来引用应用实例。这个变量在当前线程中查找活跃的应用上下文，如果找到了，它会从中获取应用实例。如果没有上下文，那么就没有办法知道哪个应用实例处于活跃状态，所以 `current_app` 就会引发一个异常。下面你可以看到它是如何在Python控制台中工作的。这需要通过运行 `python` 启动，因为 `flask shell` 命令会自动激活应用程序上下文以方便使用。

```
>>> from flask import current_app
>>> current_app.config['SQLALCHEMY_DATABASE_URI']
Traceback (most recent call last):
    ...
RuntimeError: Working outside of application context.

>>> from app import create_app
>>> app = create_app()
>>> app.app_context().push()
>>> current_app.config['SQLALCHEMY_DATABASE_URI']
'sqlite:////home/miguel/microblog/app.db'
```

这就是秘密所在！ 在调用你的视图函数之前，Flask推送一个应用上下文，它会使`current_app`和`g`生效。 当请求完成时，上下文将与这些变量一起被删除。 为了使`db.create_all()`调用在单元测试`setUp()`方法中工作，我为刚刚创建的应用程序实例推送了一个应用上下文，这样`db.create_all()`可以使用`current_app.config`知道数据库在哪里。 然后在`tearDown()`方法中，我弹出上下文以将所有内容重置为干净状态。

你还应该知道，应用上下文是Flask使用的两种上下文之一，还有一个请求上下文，它更具体，因为它适用于请求。 在处理请求之前激活请求上下文时，Flask的`request`、`session`以及Flask-Login的`current_user`变量才会变成可用状态。

环境变量

正如构建此应用时你所看到的，在启动服务器之前，有许多配置选项取决于在环境中设置的变量。 这包括密钥、电子邮件服务器信息、数据库URL和Microsoft Translator API key。 你可能会和我一样觉得，这很不方便，因为每次打开新的终端会话时，都需要重新设置这些变量。

> 译者注：可以通过将环境变量设置到开机启动中，来保持它们在该计算机中的所有终端中都生效。

应用依赖大量环境变量的常见处理模式是将这些变量存储在应用根目录中的*.env*文件中。 应用在启动时会从此文件中导入变量，这样就不需要你手动设置这些变量了。

有一个支持*.env*文件的Python包，名为`python-dotenv`。 所以让我们安装这个包：

```
(venv) $ pip install python-dotenv
```

由于*config.py*模块是我读取所有环境变量的地方，因此我将在创建Config类之前导入*.env*文件，以便在构造类时设置变量：

config.py：导入.env文件中的环境变量。

```
import os
from dotenv import load_dotenv

basedir = os.path.abspath(os.path.dirname(__file__))
load_dotenv(os.path.join(basedir, '.env'))

class Config(object):
    # ...
```

现在你可以创建一个*.env*文件并在其中写入应用所需的所有环境变量了。不要将*.env*文件加入到源代码版本控制中，这非常重要。否则，一旦你的密码和其他重要信息上传到远程代码库中后，你就会后悔莫及。

*.env*文件可以用于所有配置变量，但是不能用于Flask命令行的`FLASK_APP`和`FLASK_DEBUG`环境变量，因为它们在应用启动的早期（应用实例和配置对象存在之前）就被使用了。

以下示例显示了*.env*文件，该文件定义了一个安全密钥，将电子邮件配置为在本地运行的邮件服务器的25端口上，并且不进行身份验证，设置Microsoft Translator API key，使用数据库配置的默认值：

```
SECRET_KEY=a-really-long-and-unique-key-that-nobody-knows
MAIL_SERVER=localhost
MAIL_PORT=25
MS_TRANSLATOR_KEY=<your-translator-key-here>
```

依赖文件

此时我已经在Python虚拟环境中安装了一定数量的软件包。 如果你需要在另一台机器上重新生成你的环境，将无法记住你必须安装哪些软件包，所以一般公认的做法是在项目的根目录中写一个*requirements.txt*文件，列出所有依赖的包及其版本。 生成这个列表实际上很简单：

```
(venv) $ pip freeze > requirements.txt
```

`pip freeze`命令将安装在虚拟环境中的所有软件包以正确的格式输入到*requirements.txt*文件中。 现在，如果你需要在另一台计算机上创建相同的虚拟环境，无需逐个安装软件包，可以直接运行一条命令实现：

```
(venv) $ pip install -r requirements.txt
```

本文翻译自The Flask Mega-Tutorial Part XVI: Full-Text Search

这是Flask Mega-Tutorial系列的第十六部分，我将在其中为Microblog添加全文搜索功能。

本章的目标是为Microblog实现搜索功能，以便用户可以使用自然语言查找有趣的用户动态内容。许多不同类型的网站，都可以使用Google，Bing等搜索引擎来索引所有内容，并通过其搜索API提供搜索结果。 这这方法适用于静态页面较多的的大部分网站，比如论坛。 但在我的应用中，基本的内容单元是一条用户动态，它是整个网页的很小一部分。 我想要的搜索结果的类型是针对这些单独的用户动态而不是整个页面。 例如，如果我搜索单词“dog”，我想查看任何用户发表的包含该单词的动态。 很明显，显示所有包含“dog”（或任何其他可能的搜索字词）的用户动态的页面并不存在，大型搜索引擎也就无法索引到它。所以，我别无选择，只能自己实现搜索功能。

本章的*GitHub*链接为：*Browse*, *Zip*, *Diff.*

全文搜索引擎简介

对于全文搜索的支持不像关系数据库那样是标准化的。 有几种开源的全文搜索引擎：Elasticsearch，Apache Solr，Whoosh，Xapian，Sphinx等等，如果这还不够，常用的数据库也可以像我上面列举的那些专用搜索引擎一样提供搜索服务。 SQLite，MySQL和PostgreSQL都提供了对搜索文本的支持，以及MongoDB和CouchDB等NoSQL数据库当然也提供这样的功能。

如果你想知道哪些应用程序可以在Flask应用中运行，那么答案就是所有！ 这是Flask的强项之一，它在完成工作的同时不会自作主张。 那么到底选择哪一个呢？

在专用搜索引擎列表中，Elasticsearch非常流行，部分原因是它在ELK栈中是用于索引日志的“E”，另两个是Logstash和Kibana。 使用某个关系数据库的搜索能力也是一个不错的选择，但考虑到SQLAlchemy不支持这种功能，我将不得不使用原始SQL语句来处理搜索，否则就需要一个包， 它提供一个文本搜索的高级接口，并与SQLAlchemy共存。

基于上述分析，我将使用Elasticsearch，但我将以一种非常容易切换到另一个搜索引擎的方式来实现所有文本索引和搜索功能。 你可以用其他搜索引擎的替代替换我的实现，只需在单个模块中重写一些函数即可。

安装Elasticsearch

有几种方法可以安装Elasticsearch，包括一键安装程序，带有需要自行安装的二进制程序的zip包，甚至是Docker镜像。 该文档有一个安装页面，其中包含所有这些安装选项的详细信息。 如果你使用Linux，你可能会有一个可用于你的发行版的软件包。 如果你使用的是Mac并安装了Homebrew，那么你可以简单地运行 `brew install elasticsearch` 。

在计算机上安装Elasticsearch后，你可以在浏览器的地址栏中输入 `http://localhost:9200` 来验证它是否正在运行，预期的返回结果是JSON格式的服务基本信息。

由于我使用Python来管理Elasticsearch，因此我会使用其对应的Python客户端库：

```
(venv) $ pip install elasticsearch
```

当然不要忘记更新*requirements.txt*文件：

```
(venv) $ pip freeze > requirements.txt
```

Elasticsearch入门

我将在Python shell中为你展示使用Elasticsearch的基础知识。 这将帮助你熟悉这项服务，以便了解稍后将讨论的实现部分。

要建立与Elasticsearch的连接，需要创建一个 `Elasticsearch` 类的实例，并将连接URL作为参数传递：

```
>>> from elasticsearch import Elasticsearch
>>> es = Elasticsearch('http://localhost:9200')
```

Elasticsearch中的数据需要被写入索引中。 与关系数据库不同，数据只是一个JSON对象。 以下示例将一个包含 `text` 字段的对象写入名为 `my_index` 的索引：

```
>>> es.index(index='my_index', doc_type='my_index', id=1, body={'text': 'this is a tes
t'})
```

如果需要，索引可以存储不同类型的文档，在本处，可以根据不同的格式将 `doc_type` 参数设置为不同的值。 我要将所有文档存储为相同的格式，因此我将文档类型设置为索引名称。

对于存储的每个文档，Elasticsearch使用了一个唯一的ID来索引含有数据的JSON对象。

让我们在这个索引上存储第二个文档：

```
>>> es.index(index='my_index', doc_type='my_index', id=2, body={'text': 'a second test
'})
```

现在，该索引中有两个文档，我可以发布自由格式的搜索。 在本例中，我要搜索 `this test` ：

```
>>> es.search(index='my_index', doc_type='my_index',
... body={'query': {'match': {'text': 'this test'}}})
```

来自 `es.search()` 调用的响应是一个包含搜索结果的Python字典：

```
{
    'took': 1,
    'timed_out': False,
    '_shards': {'total': 5, 'successful': 5, 'skipped': 0, 'failed': 0},
    'hits': {
        'total': 2,
        'max_score': 0.5753642,
        'hits': [
            {
                '_index': 'my_index',
                '_type': 'my_index',
                '_id': '1',
                '_score': 0.5753642,
                '_source': {'text': 'this is a test'}
            },
            {
                '_index': 'my_index',
                '_type': 'my_index',
                '_id': '2',
                '_score': 0.25316024,
                '_source': {'text': 'a second test'}
            }
        ]
    }
}
```

在结果中你可以看到搜索返回了两个文档，每个文档都有一个分配的分数。 分数最高的文档包含我搜索的两个单词，而另一个文档只包含一个单词。 你可以看到，即使是最好的结果的分数也不是很高，因为这些单词与文本不是完全一致的。

现在，如果我搜索单词 `second` ，结果如下：

```
>>> es.search(index='my_index', doc_type='my_index',
... body={'query': {'match': {'text': 'second'}}})
{
    'took': 1,
    'timed_out': False,
    '_shards': {'total': 5, 'successful': 5, 'skipped': 0, 'failed': 0},
    'hits': {
        'total': 1,
        'max_score': 0.25316024,
        'hits': [
            {
                '_index': 'my_index',
                '_type': 'my_index',
                '_id': '2',
                '_score': 0.25316024,
                '_source': {'text': 'a second test'}
            }
        ]
    }
}
```

我仍然得到相当低的分数，因为我的搜索与文档中的文本不匹配，但由于这两个文档中只有一个包含“second”这个词，所以不匹配的根本不显示。

Elasticsearch查询对象有更多的选项，并且很好地进行了文档化，其中包含诸如分页和排序这样的和关系数据库一样的功能。

随意为此索引添加更多条目并尝试不同的搜索。 完成试验后，可以使用以下命令删除索引：

```
>>> es.indices.delete('my_index')
```

Elasticsearch配置

将Elasticsearch集成到本应用是展现Flask魅力的绝佳范例。 这是一个与Flask没有任何关系的服务和Python包，然而，我将从配置开始将它们恰如其分地集成，我先在`app.config`模块中实现这样的操作：

config.py：Elasticsearch 配置。

```
class Config(object):
    # ...
    ELASTICSEARCH_URL = os.environ.get('ELASTICSEARCH_URL')
```

与许多其他配置条目一样，Elasticsearch的连接URL将来自环境变量。 如果变量未定义，我将设置其为`None`，并将其用作禁用Elasticsearch的信号。 这主要是为了方便起见，所以当你运行应用时，尤其是在运行单元测试时，不必强制Elasticsearch服务启动和运行。 因此，为了确保服务的可用性，我需要直接在终端中定义`ELASTICSEARCH_URL`环境变量，或者将它添加到 *.env* 文件中，如下所示：

```
ELASTICSEARCH_URL=http://localhost:9200
```

使用Elasticsearch面临着非Flask插件如何使用的挑战。 我不能像在上面的例子中那样在全局范围内创建Elasticsearch实例，因为要初始化它，我需要访问`app.config`，它必须在调用`create_app()`函数后才可用。 所以我决定在应用程序工厂函数中为`app`实例添加一个`elasticsearch`属性：

`app/__init__.py`：Elasticsearch实例。

```
# ...
from elasticsearch import Elasticsearch

# ...

def create_app(config_class=Config):
    app = Flask(__name__)
    app.config.from_object(config_class)

    # ...
    app.elasticsearch = Elasticsearch([app.config['ELASTICSEARCH_URL']]) \
        if app.config['ELASTICSEARCH_URL'] else None

    # ...
```

为app实例添加一个新属性可能看起来有点奇怪，但是Python对象在结构上并不严格，可以随时添加新属性。你也可以考虑另一种方法，就是定义一个从 `Flask` 派生的子类（可以叫 `Microblog` ），然后在它的 `__init__()` 函数中定义 `elasticsearch` 属性。

请留意我设计的条件表达式，如果Elasticsearch服务的URL在环境变量中未定义，则赋值 `None` 给 `app.elasticsearch` 。

全文搜索抽象化

正如我在本章的介绍中所说的，我希望能够轻松地从Elasticsearch切换到其他搜索引擎，并且我也不希望将此功能专门用于搜索用户动态，我更愿意设计一个可复用的解决方案，如果需要，我可以轻松扩展到其他模型。 出于所有这些原因，我决定将搜索功能抽象化。 我的想法是以通用条件来设计特性，所以不会假设 `Post` 模型是唯一需要编制索引的模型，也不会假设Elasticsearch是唯一选择的搜索引擎。 但是如果我不能对任何事情做出任何假设，我是不可能完成这项工作的！

我需要的做的第一件事，是找到一种通用的方式来指定哪个模型以及其中的某个或某些字段将被索引。 我设定任何需要索引的模型都需要定义一个 `__searchable__` 属性，它列出了需要包含在索引中的字段。 对于Post模型来说，变化如下：

app/models.py: 为Post模型添加一个__searchable__属性。

```
class Post(db.Model):
    __searchable__ = ['body']
    # ...
```

需要说明的是，这个模型需要有 `body` 字段才能被索引。 不过，为了清楚地确保这一点，我添加的这个 `__searchable__` 属性只是一个变量，它没有任何关联的行为。 它只会帮助我以通用的方式编写索引函数。

我将在*app/search.py*模块中编写与Elasticsearch索引交互的所有代码。 这么做是为了将所有Elasticsearch代码限制在这个模块中。 应用的其余部分将使用这个新模块中的函数来访问索引，而不会直接访问Elasticsearch。 这很重要，因为如果有一天我不再喜欢Elasticsearch并想切换到其他引擎，我所需要做的就是重写这个模块中的函数，而应用将继续像以前一样工作。

对于本应用，我需要三个与文本索引相关的支持功能：我需要将条目添加到全文索引中，我需要从索引中删除条目（假设有一天我会支持删除用户动态），还有就是我需要执行搜索查询。 下面是*app/search.py*模块，它使用我在Python控制台中向你展示的功能实现Elasticsearch的这三个函数：

app/search.py: Search functions.

```
from flask import current_app

def add_to_index(index, model):
    if not current_app.elasticsearch:
        return
    payload = {}
    for field in model.__searchable__:
        payload[field] = getattr(model, field)
    current_app.elasticsearch.index(index=index, doc_type=index, id=model.id,
                                    body=payload)

def remove_from_index(index, model):
    if not current_app.elasticsearch:
        return
    current_app.elasticsearch.delete(index=index, doc_type=index, id=model.id)

def query_index(index, query, page, per_page):
    if not current_app.elasticsearch:
        return [], 0
    search = current_app.elasticsearch.search(
        index=index, doc_type=index,
        body={'query': {'multi_match': {'query': query, 'fields': ['*']}},
              'from': (page - 1) * per_page, 'size': per_page})
    ids = [int(hit['_id']) for hit in search['hits']['hits']]
    return ids, search['hits']['total']
```

这些函数都是通过检查 `app.elasticsearch` 是否为 `None` 开始的，如果是 `None` ，则不做任何事情就返回。 当Elasticsearch服务器未配置时，应用会在没有搜索功能的状态下继续运行，不会出现任何错误。 这都是为了方便开发或运行单元测试。

这些函数接受索引名称作为参数。 在传递给Elasticsearch的所有调用中，我不仅将这个名称用作索引名称，还将其用作文档类型，一如我在Python控制台示例中所做的那样。

添加和删除索引条目的函数将SQLAlchemy模型作为第二个参数。 `add_to_index()` 函数使用我添加到模型中的 `__searchable__` 变量来构建插入到索引中的文档。 回顾一下，Elasticsearch文档还需要一个唯一的标识符。 为此，我使用SQLAlchemy模型的 `id` 字段，该字段正好是唯一的。 在SQLAlchemy和Elasticsearch使用相同的 `id` 值在运行搜索时非常有用，因为它允许我链接两个数据库中的条目。 我之前没有提到的一点是，如果你尝试添加一个带有现有id的条目，那么Elasticsearch会用新的条目替换旧条目，所以 `add_to_index()` 可以用于新建和修改对象。

在 `remove_from_index()` 中的 `es.delete()` 函数，我之前没有展示过。 这个函数删除存储在给定 `id` 下的文档。 下面是使用相同 `id` 链接两个数据库中条目的便利性的一个很好的例子。

`query_index()` 函数使用索引名称和文本进行搜索，通过分页控件，还可以像Flask-SQLAlchemy结果那样对搜索结果进行分页。 你已经从Python控制台中看到了 `es.search()` 函数的示例用法。 我在这里发布的调用非常相似，但不是使用 `match` 查询类型，而是使用 `multi_match` ，它可以跨多个字段进行搜索。 通过传递 `*` 的字段名称，我告诉Elasticsearch查看所有字段，所以基本上我就是搜索了整个索引。 这对于使该函数具有通用性很有用，因为不同的模型在索引中可以具有不同的字段名称。

`es.search()`查询的`body`参数还包含分页参数。 `from`和`size`参数控制整个结果集的哪些子集需要被返回。 Elasticsearch没有像Flask-SQLAlchemy那样提供一个很好的Pagination对象，所以我必须使用分页数学逻辑来计算`from`值。

`query_index()`函数中的`return`语句有点复杂。 它返回两个值：第一个是搜索结果的`id`元素列表，第二个是结果总数。 两者都从`es.search()`函数返回的Python字典中获得。 用于获取ID列表的表达式，被称为列表推导式，是Python语言的一个奇妙功能，它允许你将列表从一种格式转换为另一种格式。 在本例，我使用列表推导式从Elasticsearch提供的更大的结果列表中提取`id`值。

这样看起来是否太混乱？ 也许从Python控制台演示这些函数可以帮助你更好地理解它们。 在接下来的会话中，我手动将数据库中的所有用户动态添加到Elasticsearch索引。 在我的测试数据库中，我有几条用户动态中包含数字“one”，“two”， “three”， “four” 和“five”，因此我将其用作搜索查询。 你可能需要调整你的查询以匹配数据库的内容：

```
>>> from app.search import add_to_index, remove_from_index, query_index
>>> for post in Post.query.all():
...     add_to_index('posts', post)
>>> query_index('posts', 'one two three four five', 1, 100)
([15, 13, 12, 4, 11, 8, 14], 7)
>>> query_index('posts', 'one two three four five', 1, 3)
([15, 13, 12], 7)
>>> query_index('posts', 'one two three four five', 2, 3)
([4, 11, 8], 7)
>>> query_index('posts', 'one two three four five', 3, 3)
([14], 7)
```

我发出的查询返回了七个结果。 当我以每页100项查询第1页时，我得到了全部的七项，但接下来的三个例子显示了我如何以与Flask-SQLAlchemy类似的方式对结果进行分页，当然，结果是ID列表而不是SQLAlchemy对象。

如果你想保持数据的清洁，可以在做实验之后删除`posts`索引：

```
>>> app.elasticsearch.indices.delete('posts')
```

集成SQLAlchemy到搜索

我在前面的章节中给出的解决方案是可行的，但它仍然存在一些问题。 最明显的问题是结果是以数字ID列表的形式出现的。 这非常不方便，我需要SQLAlchemy模型，以便我可以将它们传递给模板进行渲染，并且我需要用数据库中相应模型替换数字列表的方法。 第二个问题是，这个解决方案需要应用在添加或删除用户动态时明确地发出对应的索引调用，这并非不可行，但并不理想，因为在SQLAlchemy侧进行更改时错过索引调用的情况是不容易被检测到的，每当发生这种情况时，两个数据库就会越来越不同步，并且你可能在一段时间内都不会注意到。 更好的解决方案是在SQLAlchemy数据库进行更改时自动触发这些调用。

用对象替换ID的问题可以通过创建一个从数据库读取这些对象的SQLAlchemy查询来解决。这在实践中听起来很容易，但是使用单个查询来高效地实现它实际上有点棘手。

对于自动触发索引更改的问题，我决定用SQLAlchemy 事件驱动Elasticsearch索引的更新。SQLAlchemy提供了大量的事件，可以通知应用程序。 例如，每次提交会话时，我都可以定义一个由SQLAlchemy调用的函数，并且在该函数中，我可以将SQLAlchemy会话中的更新应用于Elasticsearch索引。

为了实现这两个问题的解决方案，我将编写*mixin*类。 记得mixin类吗？ 在第五章中，我将Flask-Login中的UserMixin类添加到了 `User` 模型，为它提供Flask-Login所需的一些功能。 对于搜索支持，我将定义我自己的 `SearchableMixin` 类，当它被添加到模型时，可以自动管理与SQLAlchemy模型关联的全文索引。 mixin类将充当SQLAlchemy和Elasticsearch世界之间的“粘合”层，为我上面提到的两个问题提供解决方案。

让我先告诉你实现，然后再来回顾一些有趣的细节。 请注意，这使用了多种先进技术，因此你需要仔细研究此代码以充分理解它。

app/models.py：SearchableMixin类。

```
from app.search import add_to_index, remove_from_index, query_index

class SearchableMixin(object):
    @classmethod
    def search(cls, expression, page, per_page):
        ids, total = query_index(cls.__tablename__, expression, page, per_page)
        if total == 0:
            return cls.query.filter_by(id=0), 0
        when = []
        for i in range(len(ids)):
            when.append((ids[i], i))
        return cls.query.filter(cls.id.in_(ids)).order_by(
            db.case(when, value=cls.id)), total

    @classmethod
    def before_commit(cls, session):
        session._changes = {
            'add': [obj for obj in session.new if isinstance(obj, cls)],
            'update': [obj for obj in session.dirty if isinstance(obj, cls)],
            'delete': [obj for obj in session.deleted if isinstance(obj, cls)]
        }

    @classmethod
    def after_commit(cls, session):
        for obj in session._changes['add']:
            add_to_index(cls.__tablename__, obj)
        for obj in session._changes['update']:
            add_to_index(cls.__tablename__, obj)
        for obj in session._changes['delete']:
            remove_from_index(cls.__tablename__, obj)
        session._changes = None

    @classmethod
    def reindex(cls):
        for obj in cls.query:
            add_to_index(cls.__tablename__, obj)
```

这个mixin类有四个函数，都是类方法。复习一下，类方法是与类相关联的特殊方法，而不是实例的。 请注意，我将常规实例方法中使用的 `self` 参数重命名为 `cls` ，以明确此方法接收的是类而不是实例作为其第一个参数。 例如，一旦连接到 `Post` 模型，上面的 `search()` 方法将被调用为 `Post.search()` ，而不必将其实例化。

`search()` 类方法封装来自*app/search.py*的 `query_index()` 函数以将对象ID列表替换成实例对象。你可以看到这个函数做的第一件事就是调用 `query_index()` ，并传递 `cls .__tablename__` 作为索引名称。这将是一个约定，所有索引都将用Flask-SQLAlchemy模型关联的表名。该函数返回结果ID列表和结果总数。通过它们的ID检索对象列表的SQLAlchemy查询基于SQL语言的 `CASE` 语句，该语句需要用于确保数据库中的结果与给定ID的顺序相同。这很重要，因为Elasticsearch查询返回的结果不是有序的。如果你想了解更多关于这个查询的工作方式，你可以参考这个StackOverflow问题的接受答案。 `search()` 函数返回替换ID列表的查询结果集，以及搜索结果的总数。

`before_commit()` 和 `after_commit()` 方法分别对应来自SQLAlchemy的两个事件，这两个事件分别在提交发生之前和之后触发。 前置处理功能很有用，因为会话还没有提交，所以我可以查看并找出将要添加，修改和删除的对象，如 `session.new` ， `session.dirty` 和 `session.deleted` 。 这些对象在会话提交后不再可用，所以我需要在提交之前保存它们。 我使用 `session._changes` 字典将这些对象写入会话提交后仍然存在的地方，因为一旦会话被提交，我将使用它们来更新Elasticsearch索引。

当调用 `after_commit()` 处理程序时，会话已成功提交，因此这是在Elasticsearch端进行更新的适当时间。 session对象具有 `before_commit()` 中添加的_changes变量，所以现在我可以迭代需要被添加，修改和删除的对象，并对*app/search.py*中的索引函数进行相应的调用。

`reindex()` 类方法是一个简单的帮助方法，你可以使用它来刷新所有数据的索引。 你看到我在上面做的将所有用户动态初始加载到测试索引中，这个操作与Python shell会话中的类似。 有了这个方法，我可以调用 `Post.reindex()` 将数据库中的所有用户动态添加到搜索索引中。

为了将 `SearchableMixin` 类整合到 `Post` 模型中，我必须将它作为 `Post` 的基类，并且还需要监听提交之前和之后的事件：

app/models.py：添加SearchableMixin类到Post模型。

```
class Post(SearchableMixin, db.Model):
    # ...

db.event.listen(db.session, 'before_commit', Post.before_commit)
db.event.listen(db.session, 'after_commit', Post.after_commit)
```

请注意， `db.event.listen()` 调用不在类内部，而是在其后面。 这两行代码设置了每次提交之前和之后调用的事件处理程序。 现在 `Post` 模型会自动为用户动态维护一个全文搜索索引。 我可以使用 `reindex()` 方法来初始化当前在数据库中的所有用户动态的索引：

```
>>> Post.reindex()
```

我可以通过运行`Post.search()`来搜索使用SQLAlchemy模型的用户动态。 在下面的例子中，我要求查询第一页的五个元素：

```
>>> query, total = Post.search('one two three four five', 1, 5)
>>> total
7
>>> query.all()
[<Post five>, <Post two>, <Post one>, <Post one more>, <Post one>]
```

搜索表单

的确有些激进。 我上面做的保持通用性的工作涉及到几个高级主题，因此可能需要一些时间才能完全理解。 现在我有一套完整的系统来处理用户动态的自然语言搜索。 所以现在需要做的是将所有这些功能与应用集成在一起。

基于网络搜索的一种相当标准的方法是在URL的查询字符串中将搜索词作为`q`参数的值。 例如，如果你想在Google上搜索`Python`，并且想要节约少许时间，则只需在浏览器的地址栏中输入以下URL即可直接查看结果：

```
https://www.google.com/search?q=python
```

允许将搜索完全封装在URL中是很好的，因为这方便了与其他人共享，只要点击链接就可以访问搜索结果。

请允许我向你介绍一种区别于以前的Web表单的处理方式。 我曾经使用`POST`请求来提交表单数据，但是为了实现上述搜索，表单提交必须以`GET`请求发送，这是一种请求方法，当你在浏览器中输入网址或点击链接时，就是`GET`请求。 另一个有趣的区别是搜索表单将存在于导航栏中，因此它将会出现应用的所有页面中。

这里是搜索表单类，只有`q`文本字段：

app/main/forms.py：搜索表单。

```
from flask import request

class SearchForm(FlaskForm):
    q = StringField(_l('Search'), validators=[DataRequired()])

    def __init__(self, *args, **kwargs):
        if 'formdata' not in kwargs:
            kwargs['formdata'] = request.args
        if 'csrf_enabled' not in kwargs:
            kwargs['csrf_enabled'] = False
        super(SearchForm, self).__init__(*args, **kwargs)
```

`q`字段不需要任何解释，因为它与我以前使用的其他文本字段相似。在这个表单中，我不需要提交按钮。对于具有文本字段的表单，当焦点位于该字段上时，你按下Enter键，浏览器将提交表单，因此不需要按钮。我还添加了一个`__init__`构造函数，它提供

了 `formdata` 和 `csrf_enabled` 参数的值（如果调用者没有提供它们的话）。 `formdata` 参数决定Flask-WTF从哪里获取表单提交。缺省情况是使用 `request.form` ，这是Flask放置通过 `POST` 请求提交的表单值的地方。通过 `GET` 请求提交的表单在查询字符串中传递字段值，所以我需要将Flask-WTF指向 `request.args` ，这是Flask写查询字符串参数的地方。你是否还记得的，表单默认添加了CSRF保护，包含一个CSRF标记，该标记通过模板中的 `form.hidden_tag()` 构造添加到表单中。为了使搜索表单运作，CSRF需要被禁用，所以我将 `csrf_enabled` 设置为 `False` ，以便Flask-WTF知道它需要忽略此表单的CSRF验证。

由于我需要在所有页面中都显示此表单，因此无论用户在查看哪个页面，我都需要创建一个 `SearchForm` 类的实例。 唯一的要求是用户登录，因为对于匿名用户，我目前不会显示任何内容。 与其在每个路由中创建表单对象，然后将表单传递给所有模板，我将向你展示一个非常有用的技巧，当你需要在整个应用中实现一个功能时，可以消除重复代码。 回到第六章，我已经使用了 `before_request` 处理程序， 来记录每个用户上次访问的时间。 我要做的是在同样的功能中创建我的搜索表单，但有一点区别：

app/main/routes.py：在请求处理前的处理器中初始化搜索表单。

```
from flask import g
from app.main.forms import SearchForm

@bp.before_app_request
def before_request():
    if current_user.is_authenticated:
        current_user.last_seen = datetime.utcnow()
        db.session.commit()
        g.search_form = SearchForm()
    g.locale = str(get_locale())
```

在这里，当用户已认证时，我会创建一个搜索表单类的实例。当然，我需要这个表单对象一直存在，直到它可以在请求结束时渲染，所以我需要将它存储在某个地方。那个地方就是Flask提供的 `g` 容器。这个 `g` 变量是应用可以存储需要在整个请求期间持续存在的数据的地方。在这里，我将表单存储在 `g.search_form` 中，所以当请求前置处理程序结束并且Flask调用处理请求的URL的视图函数时， `g` 对象将会是相同的，并且表单仍然存在。请注意，这个 `g` 变量对每个请求和每个客户端都是特定的，因此即使你的Web服务器一次为不同的客户端处理多个请求，仍然可以依靠 `g` 来专用存储各个请求的对应变量。

下一步是将表单渲染成页面。 我在上面说过，我想在所有页面中展示这个表单，所以更有意义的是将其作为导航栏的一部分进行渲染。 事实上，这很简单，因为模板也可以看到存储在 `g` 变量中的数据，所以我不需要在所有 `render_template()` 调用中将表单作为显式模板参数添加进去。以下是我如何在基础模板中渲染表单的代码：

app/templates/base.html：在导航栏中渲染搜索表单。

```
            ...
            <div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">
                <ul class="nav navbar-nav">
                    ... home and explore links ...
                </ul>
                {% if g.search_form %}
                <form class="navbar-form navbar-left" method="get"
                        action="{{ url_for('main.search') }}">
                    <div class="form-group">
                        {{ g.search_form.q(size=20, class='form-control',
                            placeholder=g.search_form.q.label.text) }}
                    </div>
                </form>
                {% endif %}
                ...
```

只有在定义了 `g.search_form` 时才会渲染表单。 此检查是必要的，因为某些页面（如错误页面）可能没有定义它。 这个表单与我之前做过的略有不同。 我将 `method` 属性设置为 `get` ，因为我希望表单数据作为查询字符串，通过 `GET` 请求提交。 另外，我创建的其他表单 `action` 属性为空，因为它们被提交到渲染表单的同一页面。 而这个表单很特殊，因为它出现在所有页面中，所以我需要明确告诉它需要提交的地方，这是专门用于处理搜索的新路由。

搜索视图函数

完成搜索功能的最后一项功能是接收搜索表单的视图函数。 该视图函数将被附加到*/search*路由，以便你可以发送类似*http://localhost:5000/search?q=search-words*的搜索请求，就像Google一样。

app/main/routes.py：搜索视图函数。

```
@bp.route('/search')
@login_required
def search():
    if not g.search_form.validate():
        return redirect(url_for('main.explore'))
    page = request.args.get('page', 1, type=int)
    posts, total = Post.search(g.search_form.q.data, page,
                               current_app.config['POSTS_PER_PAGE'])
    next_url = url_for('main.search', q=g.search_form.q.data, page=page + 1) \
        if total > page * current_app.config['POSTS_PER_PAGE'] else None
    prev_url = url_for('main.search', q=g.search_form.q.data, page=page - 1) \
        if page > 1 else None
    return render_template('search.html', title=_('Search'), posts=posts,
                           next_url=next_url, prev_url=prev_url)
```

你已经看到，在其他表单中，我使用 `form.validate_on_submit()` 方法来检查表单提交是否有效。 不幸的是，该方法只适用于通过 `POST` 请求提交的表单，所以对于这个表单，我需要使用 `form.validate()` ，它只验证字段值，而不检查数据是如何提交的。 如果验证失败，这是因为用户提交了一个空的搜索表单，所以在这种情况下，我只能重定向到了显示所有用户动态的发现页面。

`SearchableMixin`类中的`Post.search()`方法用于获取搜索结果列表。 分页的处理方式与主页和发现页面非常类似，但如果没有Flask-SQLAlchemy的“分页”对象的帮助，生成下一个和前一个链接会有点棘手。 这是从`Post.search()`返回的结果总数的用途所在。

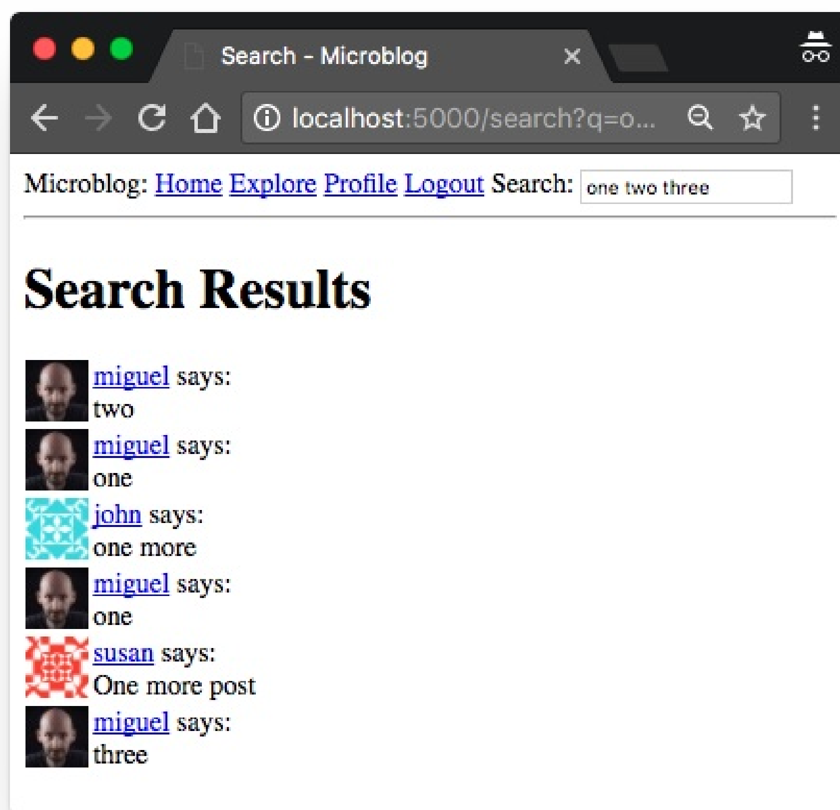
一旦计算出搜索结果和分页链接的页面，剩下的就是渲染一个包含所有这些数据的模板。 我已经想出了一种重用*index.html*模板来显示搜索结果的方法，但考虑到有一些差异，我决定创建一个专用于显示搜索结果的*search.html*专属模板， 以*_post.html*子模板的优势来渲染搜索结果：

app/templates/search.html：搜索结果模板。

```
{% extends "base.html" %}

{% block app_content %}
    <h1>{{ _('Search Results') }}</h1>
    {% for post in posts %}
        {% include '_post.html' %}
    {% endfor %}
    <nav aria-label="...">
        <ul class="pager">
            <li class="previous{% if not prev_url %} disabled{% endif %}">
                <a href="{{ prev_url or '#' }}">
                    <span aria-hidden="true">&larr;</span>
                    {{ _('Previous results') }}
                </a>
            </li>
            <li class="next{% if not next_url %} disabled{% endif %}">
                <a href="{{ next_url or '#' }}">
                    {{ _('Next results') }}
                    <span aria-hidden="true">&rarr;</span>
                </a>
            </li>
        </ul>
    </nav>
{% endblock %}
```

如果前一个和下一个链接的渲染逻辑有点混乱，可能查看分页组件的Bootstrap文档会有所帮助。



感想如何？ 本章的内容有些激进，因为里面介绍了一些相当先进的技术。 本章中的一些概念可能需要你花一些时间才能有所领悟。本章最重要的一点是，如果你想使用与Elasticsearch不同的搜索引擎，只需要重写*app/search.py*即可。 通过这项工作的另一个重要好处是，如果我需要为另外的数据库模型添加搜索支持，我可以简单地通过向它添加 `SearchableMixin` 类，为 `__searchable__` 属性填写要索引的字段列表和SQLAlchemy事件处理程序的监听即可。 我认为这些努力是值得的，因为从现在起，处理全文索引将会变得十分容易。

本文翻译自The Flask Mega-Tutorial Part XVII: Deployment on Linux

这是Flask Mega-Tutorial系列的第十七部分，我将把Microblog部署到Linux服务器。

在本章中，我将谈到Microblog应用生命周期中的一个里程碑，因为我将讨论如何将应用部署到生产服务器上，以便真实用户可以访问它。

部署的主题非常广泛，因此不可能在这里涵盖所有范畴。 本章致力于探讨传统托管方式，包括Ubuntu发行版的Linux服务器和流行的树莓派微机。 我将在后面的章节中介绍其他选项，例如云和容器部署。

本章的*GitHub*链接为：*Browse*, *Zip*, *Diff.*

传统托管

当提到“传统托管”时，意思是应用是手动或通过原始服务器机器上的脚本安装部署的。 该过程涉及安装应用程序、其依赖项和生产规模的Web服务器，并配置系统以确保其安全。

当你要部署自己的项目时，要问的第一个问题是在哪找服务器。 目前有很多经济的托管服务。 只需每月5美元，Digital Ocean，Linode或Amazon Lightsail就可以租借一台虚拟化Linux服务器（Linode和Digital Ocean为其入门级服务器提供1GB RAM，而亚马逊仅提供512MB）给你运行部署实验。 如果你一分钱都不愿意花，那么Vagrant和VirtualBox组合而成的工具，可以让你在自己的计算机上创建一个与付费服务器类似的虚拟服务器。

就技术角度而言，该应用可以部署在任何主流操作系统上，包括各种开放源代码的Linux和BSD发行版以及商用的OS X（OS X是一个开源和商业的混种，因为它基于开源BSD衍生产品Darwin）和Microsoft Windows。

由于OS X和Windows是的桌面操作系统，不是作为服务器的最佳选择，因此不是首选。Linux或BSD操作系统之间的选择很大程度上取决于爱好，所以我将选择其中更受欢迎的Linux。 而Linux发行版中，我将再次选择受欢迎的Ubuntu。

创建Ubuntu服务器

如果你有兴趣与我一起部署，那么就需要一台服务器才能开始工作。 为你推荐两种选择，一种是付费的，另一种是免费的。 如果你愿意花一点钱，可以在Digital Ocean，Linode或Amazon Lightsail上注册一个账户，并创建一个Ubuntu 16.04镜像的虚拟服务器。 你应该使用最低配置的服务器，在我写这篇文章的时候，三家的最低配置都是每月5美元。 开销是按照服务器启动的小时数进行比例计算的，因此，如果你创建服务器后，使用几个小时然后删除它，那么有可能你只需支付美分级别的费用。

免费的方案基于你的计算机上可以运行虚拟机。 要使用此选项，请在你的机器上安装Vagrant和VirtualBox，然后创建一个名为*Vagrantfile*的文件并用以下内容来描述虚拟机的规格：

Vagrantfile：Vagrant配置。

```
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/xenial64"
  config.vm.network "private_network", ip: "192.168.33.10"
  config.vm.provider "virtualbox" do |vb|
    vb.memory = "1024"
  end
end
```

该文件配置了一个带有1GB RAM的Ubuntu 16.04服务器，你可以用其IP地址192.168.33.10来访问该服务器。 要创建服务器，请运行以下命令：

```
$ vagrant up
```

请参阅Vagrant 命令行文档了解其他管理虚拟服务器的选项。

使用SSH客户端

你的服务器处于后端，所以不需要像个人计算机上那样拥有桌面。 你可以通过SSH客户端连接到服务器，并运行命令行进行交互。 如果你使用的是Linux或Mac OS X，则可能已经安装了OpenSSH。 如果你使用Microsoft Windows，Cygwin，Git和Windows Subsystem for Linux提供OpenSSH，因此你可以安装这些选项中的任何一个。

如果你正在使用来自第三方提供商的虚拟服务器，则在创建服务器时，会为其分配IP地址。 你可以使用以下命令打开终端会话来连接到该服务器：

```
$ ssh root@<server-ip-address>
```

系统会提示你输入密码。密码已在创建服务器后自动生成并显示给你，或者你自己指定了密码。

如果你使用的是Vagrant VM，则可以使用以下命令打开终端会话：

```
$ vagrant ssh
```

如果你使用的是Windows并且拥有Vagrant虚拟机，请注意你需要从可以调用 `ssh` 命令的shell运行上述命令。

免密登录

如果你使用的是Vagrant虚拟机，那么可以跳过本节，因为你的虚拟机已正确配置为使用名为 `ubuntu` 的非root帐户，Vagrant不用输入密码就可以自动登录。

要是你使用的是虚拟服务器，则建议创建一个常规用户来完成你的部署工作，并配置此帐户以便在不使用密码的情况下登录，这么做最初看起来似乎是一个糟糕的主意， 之后你会发现它不仅更方便，而且更安全。

我将创建一个名为`ubuntu`的用户帐户（如果你愿意，可以使用其他名称）。 要创建这个用户，请使用前一节中的`ssh`指令登录到你的服务器的root帐户，然后键入以下命令来创建用户，给它`sudo`权限并最终切换到它：

```
$ adduser --gecos "" ubuntu
$ usermod -aG sudo ubuntu
$ su ubuntu
```

现在我要配置这个新的`ubuntu`帐户来使用public key认证，以便你可以免密登录。

先不管服务器上打开的终端会话，然后在本地计算机上启动第二个终端。 如果你使用的是Windows，这需要是可以访问`ssh`命令的终端，所以它可能是一个`bash`或者类似的提示符的终端，而不是本地的Windows终端。 在该终端会话中，检查*~/.ssh*目录的内容：

```
$ ls ~/.ssh
id_rsa  id_rsa.pub
```

如果目录列表显示如上所述的名为*id_rsa*和*id_rsa.pub*的文件，那么你已经有一个密钥。 如果没有这两个文件，或者根本没有*~/.ssh*目录，则你需要运行以下命令（也是OpenSSH工具集的一部分）来创建SSH密钥对：

```
$ ssh-keygen
```

此应用程序将提示你输入一些内容，为此我建议你在所有提示中按Enter以接受默认设置。 你当然也可以做一些设置，如果你知道这么做意味着什么的话。

运行此命令后，应该有上面列出的两个文件了。 文件*id_rsa.pub*是你的公钥，这是一个你将提供给第三方的文件，用于识别你的身份。 *id_rsa*文件是你的私钥，不应与任何人共享。

你现在需要将公钥配置为服务器中的授权主机。 在你自己的计算机上打开的终端上，将公钥打印到屏幕上：

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCjw....F8Xv4f/0+7WT miguel@miguelspc
```

这将是一个非常长的字符序列，显示时可能跨越多行（但实际上只有一行）。 你需要将此数据复制到剪贴板，然后切换回远程服务器上的终端，你将在其中运行以下命令来存储公钥：

```
$ echo <paste-your-key-here> >> ~/.ssh/authorized_keys
$ chmod 600 ~/.ssh/authorized_keys
```

免密登录现在应该可以工作了。 背后逻辑是，你机器上的`ssh`会用私钥执行加密操作来向服务器标识自己。 然后服务器使用你的公钥验证操作是否有效。

你现在可以注销`ubuntu`会话，然后注销`root`会话，然后尝试直接登录到`ubuntu`帐户：

```
$ ssh ubuntu@<server-ip-address>
```

这一次不用输入密码就登录了！

保护你的服务器

为了最大限度地降低服务器受到攻击的风险，你可以采取一些措施来关闭攻击者可能访问的大量潜在漏洞。

我要做的第一个更改是禁用root用户通过SSH登录。 你现在可以无密码地访问`ubuntu`帐户，并且可以通过`sudo`从该帐户运行管理员命令，因此实际上不需要暴露root帐户。 要禁用root登录，你需要编辑服务器上的*/etc/ssh/sshd_config*文件。 你可能在你的服务器上安装了`vi`和`nano`文本编辑器，你可以用它来编辑文件（如果你不熟悉这两种文件编辑器，可以首先尝试`nano`）。 由于SSH配置对普通用户是不可访问的，所以你需要在编辑器命令前添加`sudo`（即`sudo vi /etc/ssh/sshd_config`）。 你需要更改此文件中的单行：

/etc/ssh/sshd_config：禁止root登录。

```
PermitRootLogin no
```

请注意，要进行此更改，你需要找到以`PermitRootLogin`开头的行（找不到就新建一行）并将该值更改为`no`。

下一个更改在同一个文件中。 现在我要为所有帐户禁用密码登录。 你有一个无密码的登录设置，所以没有必要允许密码。 如果你对完全禁用密码感到紧张，可以跳过此更改，但对于生产服务器来说，这是一个非常好的主意，因为攻击者经常在所有服务器上尝试随机帐户名和密码并希望能中奖。 要禁用密码登录，请在*/etc/ssh/sshd_config*中更改以下行：

/etc/ssh/sshd_config：禁用密码登录。

```
PasswordAuthentication no
```

完成编辑SSH配置后，需要重新启动ssh服务以使更改生效：

```
$ sudo service ssh restart
```

我要做的第三个改变是安装防火墙。 这是一个阻止在任何未明确启用的端口上访问服务器的软件：

```
$ sudo apt-get install -y ufw
$ sudo ufw allow ssh
$ sudo ufw allow http
$ sudo ufw allow 443/tcp
$ sudo ufw --force enable
$ sudo ufw status
```

这些命令会安装ufw（简单防火墙），并将其配置为仅允许端口22（ssh），80（http）和443（https）上的外部通信。 任何其他端口将不被允许。

安装基础依赖

如果你遵循了我的建议并配置了Ubuntu 16.04发行版的服务器，那么你的系统完全支持Python 3.5，因此这是我将用于部署的Python版本。

基础的Python解释器可能已经预先安装在你的服务器上，但有一些额外的软件包可能却没有，而且Python之外还有一些其他软件包可用于创建健壮的生产环境部署。 对于数据库服务器，我将从SQLite切换到MySQL。 Postfix包是一个邮件传输代理，我将用它来发送电子邮件。 Supervisor工具将监视Flask服务器进程，并在其崩溃时自动重启，并当Supervisor服务重启后自动启动其监视的服务。 Nginx服务器将接受来自外部世界的所有请求，并将它们转发给应用程序。 最后，我将使用git来从git仓库下载应用程序。

```
$ sudo apt-get -y update
$ sudo apt-get -y install python3 python3-venv python3-dev
$ sudo apt-get -y install mysql-server postfix supervisor nginx git
```

这些安装大部分是无人值守的，但是在运行第三条安装语句到一定进度时，系统会提示你为MySQL服务选择一个root密码，并且还会询问关于安装postfix软件包的一些问题，你可以接受他们的默认答案。

请注意，对于此部署，我选择不安装Elasticsearch。 这项服务需要大量的RAM，所以只有拥有超过2GB内存的大型服务器时才可以考虑。 为了避免服务器内存不足的问题，我将停用搜索功能。 如果你有高配的服务器，可以从Elasticsearch站点下载官方的.deb软件包，并按照其安装说明将其添加到你的服务器。 请注意，Ubuntu 16.04软件包存储库中提供的Elasticsearch软件包太旧，无法运行，你需要6.x或更高版本。

我还注意到，默认安装的postfix可能不足以在生产环境中发送电子邮件。 为了避免垃圾邮件和恶意邮件，很多服务器都要求发件人服务器通过安全扩展标识自己，这意味着至少你必须拥有与你的服务器相关联的域名。 如果你想了解如何完全配置电子邮件服务器以使其通过标准安全测试，请参阅以下Digital Ocean的指南：

- Postfix Configuration

- Adding an SPF Record
- DKIM Installation and Configuration

安装应用

现在我要使用 `git` 从我的GitHub代码库下载Microblog源代码。 如果你不熟悉git源码控制，我建议你阅读git for beginners。

要将应用下载到服务器，请确保你位于 `ubuntu` 用户的主目录中，然后运行：

```
$ git clone https://github.com/miguelgrinberg/microblog
$ cd microblog
$ git checkout v0.17
```

这会将代码克隆到你的服务器上，并将其同步到本章的内容。 如果你在学习本教程的过程中维护了自己的git代码库，则可以将代码库URL更改为你的URL，在这种情况下，你可以跳过 `git checkout` 命令。

现在我需要创建一个虚拟环境并使用所有的包依赖项来填充它，在第十五章中，我已将依赖包的列表保存到*requirements.txt*文件中：

```
$ python3 -m venv venv
$ source venv/bin/activate
(venv) $ pip install -r requirements.txt
```

除了*requirements.txt*中的包之外，我还将使用此生产部署指定的两个包，因此它们不包含在*requirements.txt*文件中。 `gunicorn` 软件包是Python应用程序的生产Web服务器。 `pymysql` 软件包包含MySQL驱动程序，它使SQLAlchemy能够与MySQL数据库一起工作：

```
(venv) $ pip install gunicorn pymysql
```

我需要创建一个*.env*文件，其中包含所有需要的环境变量：

/home/ubuntu/microblog/.env：环境配置。

```
SECRET_KEY=52cb883e323b48d78a0a36e8e951ba4a
MAIL_SERVER=localhost
MAIL_PORT=25
DATABASE_URL=mysql+pymysql://microblog:<db-password>@localhost:3306/microblog
MS_TRANSLATOR_KEY=<your-translator-key-here>
```

这个*.env*文件与我在第十五章展示的非常类似，但是我为SECRET_KEY使用了一个随机字符串。 为了生成这个随机字符串，我使用了下面的命令：

```
python3 -c "import uuid; print(uuid.uuid4().hex)
```

对于`DATABASE_URL`变量，我定义了一个MySQL URL。 我将在下一节中向你介绍如何配置数据库。

我需要将`FLASK_APP`环境变量设置为应用程序的入口点以启用`flask`命令，但在解析*.env*文件之前需要此变量，因此需要手动设置。 为避免每次都设置它，我把它添加到`ubuntu`帐户的*~/.profile*文件的底部，以便每次登录时自动设置它：

```
$ echo "export FLASK_APP=microblog.py" >> ~/.profile
```

如果你注销并重新登录，现在`FLASK_APP`就已经设置好了。 你可以通过运行`flask --help`来确认它是否已经设置好了。 如果帮助信息显示应用程序已添加的`translate`命令，那么你就知道应用程序已被找到。

现在`flask`命令是有效的，我可以编译语言翻译：

```
(venv) $ flask translate compile
```

设置MySQL

我在开发过程中使用过的sqlite数据库非常适合简单的应用程序，但是当部署可能需要一次处理多个请求的健壮Web服务器时，最好使用更强大的数据库。 出于这个原因，我要建立一个名为'microblog'的MySQL数据库。

要管理数据库服务器，我将使用`mysql`命令，该命令应该已经安装在你的服务器上：

```
$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 6
Server version: 5.7.19-0ubuntu0.16.04.1 (Ubuntu)

Copyright (c) 2000, 2017, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

请注意，你需要键入你在安装MySQL时选择的MySQL root密码才能访问MySQL命令提示符。

这些是创建名为`microblog`的新数据库的命令，以及具有完全访问权限的同名用户：

```
mysql> create database microblog character set utf8 collate utf8_bin;
mysql> create user 'microblog'@'localhost' identified by '<db-password>';
mysql> grant all privileges on microblog.* to 'microblog'@'localhost';
mysql> flush privileges;
mysql> quit;
```

你将需要用你选择的密码来替换`<db-password>`。 这将是`microblog`数据库用户的密码，所以不要使用你已为root用户选择的密码。 `microblog`用户的密码需要与你包含在*.env*文件中的`DATABASE_URL`变量中的密码相匹配。

如果你的数据库配置是正确的，你现在应该能够运行数据库迁移以创建所有的表：

```
(venv) $ flask db upgrade
```

继续下一步之前，确保上述命令成功完成且不会产生任何错误。

设置Gunicorn和Supervisor

当你使用`flask run`运行服务器时，正在使用的是Flask附带的Web服务器。 该服务器在开发过程中非常有用，但它不适合用于生产服务器，因为它不考虑性能和稳健性。 取而代之，我决定使用gunicorn，它是一个纯粹的Python Web服务器，但与Flask不同，它是一个支持高并发的强大生产服务器，同时它也非常容易使用。

要在gunicorn下启动Microblog，你可以使用以下命令：

```
(venv) $ gunicorn -b localhost:8000 -w 4 microblog:app
```

`-b`选项告诉gunicorn在哪里监听请求，我在8000端口上监听了内部网络接口。 在没有外部访问的情况下运行Python Web应用程序通常是一个好主意，然后还需要一个非常快速的Web服务器，它可以优化来自客户端的所有静态文件的请求。 这个快速的Web服务器将直接提供静态文件，并将用于应用程序的任何请求转发到内部服务器。 我将在下一节中向你展示如何将nginx设置为面向公众的服务器。

`-w`选项配置gunicorn将运行多少*worker*。 拥有四个进程可以让应用程序同时处理多达四个客户端，这对于Web应用程序通常足以处理大量客户端请求，因为并非所有客户端都在不断请求内容。 根据服务器的RAM大小，你可能需要调整worker数量，以免内存不足。

`microblog:app`参数告诉gunicorn如何加载应用程序实例。 冒号前的名称是包含应用程序的模块，冒号后面的名称是此应用程序的名称。

虽然gunicorn的设置非常简单，但从命令行运行服务器在生产服务器实际上不是一个恰当的方案。 我想要做的是让服务器在后台运行，并持续监视，因为如果由于某种原因导致服务器崩溃并退出，我想确保新的服务器自动启动以取代它。 而且我还想确保如果机器重新启动，服

务器在启动时自动运行，而无需人工登录和启动。 我将使用上面安装的supervisor包来执行此操作。

Supervisor使用配置文件定义它要监视什么程序以及如何在必要时重新启动它们。 配置文件必须存储在*/etc/supervisor/conf.d*中。 这是Microblog的配置文件，我将其称为*microblog.conf*：

/etc/supervisor/conf.d/microblog.conf：Supervisor配置。

```
[program:microblog]
command=/home/ubuntu/microblog/venv/bin/gunicorn -b localhost:8000 -w 4 microblog:app
directory=/home/ubuntu/microblog
user=ubuntu
autostart=true
autorestart=true
stopasgroup=true
killasgroup=true
```

`command`，`directory` 和 `user` 设置告诉supervisor如何运行应用程序。 如果计算机启动或崩溃，`autostart` 和 `autorestart` 设置会使microblog自动重新启动。 `stopasgroup` 和 `killasgroup` 选项确保当supervisor需要停止应用程序来重新启动它时，它仍然会调度成顶级gunicorn进程的子进程。

编写此配置文件后，必须重载supervisor服务的配置才能导入它：

```
$ sudo supervisorctl reload
```

像这样，这个gunicorn web服务器就已经启动和运行，并处于监控之中！

设置Nginx

由gunicorn启动的microblog应用服务器现在运行在本地端口8000。 我现在需要做的是将应用程序暴露给外部世界，为了使面向公众的web服务器能够被访问，我在防火墙上打开了两个端口（80和443）来处理应用程序的Web通信。

我希望这是一个安全的部署，所以我要配置端口80将所有流量转发到将要加密的端口443。 我将首先创建一个SSL证书。创建一个自签名*SSL*证书，这对于测试是可以的，但对于真正的部署不太好，因为Web浏览器会警告用户，证书不是由可信证书颁发机构颁发的。 创建microblog的SSL证书的命令是：

```
$ mkdir certs
$ openssl req -new -newkey rsa:4096 -days 365 -nodes -x509 \
  -keyout certs/key.pem -out certs/cert.pem
```

该命令将要求你提供关于应用程序和你自己的一些信息。 这些信息将包含在SSL证书中，如果用户请求查看它，Web浏览器则会向用户显示它们。上述命令的结果将是名为*key.pem*和*cert.pem*的两个文件，我将其放置在Microblog根目录的*certs*子目录中。

要有一个由nginx服务的网站，你需要为它编写配置文件。 在大多数nginx安装中，这个文件需要位于*/etc/nginx/sites-enabled*目录中。Nginx在这个位置安装了一个我不需要的测试站点，所以我将首先删除它：

```
$ sudo rm /etc/nginx/sites-enabled/default
```

下面你可以看到Microblog的nginx配置文件，它在*/etc/nginx/sites-enabled/microblog*中：

/etc/nginx/sites-enabled/microblog：Nginx配置。

```
server {
    # listen on port 80 (http)
    listen 80;
    server_name _;
    location / {
        # redirect any requests to the same URL but on https
        return 301 https://$host$request_uri;
    }
}
server {
    # listen on port 443 (https)
    listen 443 ssl;
    server_name _;

    # location of the self-signed SSL certificate
    ssl_certificate /home/ubuntu/microblog/certs/cert.pem;
    ssl_certificate_key /home/ubuntu/microblog/certs/key.pem;

    # write access and error logs to /var/log
    access_log /var/log/microblog_access.log;
    error_log /var/log/microblog_error.log;

    location / {
        # forward application requests to the gunicorn server
        proxy_pass http://localhost:8000;
        proxy_redirect off;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }

    location /static {
        # handle static files directly, without forwarding to the application
        alias /home/ubuntu/microblog/static;
        expires 30d;
    }
}
```

Nginx的配置不易理解，但我添加了一些注释，至少你可以知道每个部分的功能。 如果你想获得关于特定指令的信息，请参阅nginx官方文档。

添加此文件后，你需要告诉nginx重新加载配置以激活它：

```
$ sudo service nginx reload
```

现在应用程序应该部署成功了。 在你的Web浏览器中，可以键入服务器的IP地址（如果使用的是Vagrant VM，则为192.168.33.10），然后该服务器将连接到应用程序。 由于你使用的是自签名证书，因此将收到来自Web浏览器的警告，你必须解除该警告。

使用上述说明为自己的项目完成部署之后，我强烈建议你将自签名证书替换为真实的证书，以便浏览器不会在用户访问你的网站时发出警告。 为此，你首先需要购买域名并将其配置为指向你的服务器的IP地址。 一旦你有一个域名，你可以申请一个免费的Let's Encrypt SSL证书。 我在博客上写了一篇关于如何通过HTTPS运行你的Flask应用程序的详细文章。

部署应用更新

我想讨论的基于Linux的部署的最后一个主题是如何处理应用程序升级。 应用程序源代码通过 `git` 安装在服务器中，因此，无论何时想要将应用程序升级到最新版本，都可以运行 `git pull` 来下载自上次部署以来的新提交。

当然，下载新版本的代码不会导致升级。 当前正在运行的服务器进程将继续运行，旧代码已被读取并存储在内存中。 要触发升级，你必须停止当前的服务器并启动一个新的服务器，以强制重新读取所有代码。

进行升级通常比重新启动服务器更为复杂。 你可能需要应用数据库迁移或编译新的语言翻译，因此实际上，执行升级的过程涉及一系列命令：

```
(venv) $ git pull                              # download the new version
(venv) $ sudo supervisorctl stop microblog     # stop the current server
(venv) $ flask db upgrade                      # upgrade the database
(venv) $ flask translate compile               # upgrade the translations
(venv) $ sudo supervisorctl start microblog    # start a new server
```

树莓派托管

树莓派是一款革命性低成本的小型Linux计算机，功耗非常低，因此它是托管家庭在线服务器的理想设备，可以全天候在线而无需捆绑你的台式电脑或笔记本电脑。 有几个Linux发行版可以在树莓派上运行。 我的选择是Raspbian，这是树莓派基金会的官方发行版。

为了准备树莓派的环境，我要安装一个新的Raspbian版本。 我将使用2017年9月版的Raspbian Stretch Lite，但在阅读本文时，可能会有更新的版本，请查看官方下载页面获得最新版本。

Raspbian镜像需要安装在SD卡上，然后插入树莓派，以便它启动时可以识别到。 在树莓派站点上可以查看到从Windows，Mac OS X和Linux将Raspbian镜像复制到SD卡的方法。

当你第一次启动树莓派时，请在连接到键盘和显示器时进行操作，以便你可以进行设置。 至少应该启用SSH，以便你可以从计算机登录并方便地执行部署任务。

和Ubuntu一样，Raspbian也是Debian的衍生产品，所以上面针对的Ubuntu Linux的说明，大部分都可以在树莓派上生效。 但是，如果你计划在家庭网络上运行小型应用程序而无需外部访问时，则可以跳过某些步骤。 例如，你可能不需要防火墙或无密码登录。 你可能想在这样一台小型的计算机上使用SQLite而不是MySQL。 你可以选择不使用nginx，并且让gunicorn服务器直接监听来自客户端的请求。 你可能只想要一个gunicorn worker进程。 Supervisor服务对于确保应用程序始终处于运行状态非常有用，因此我建议你仍然在树莓派上使用它。

本文翻译自The Flask Mega-Tutorial Part XVIII: Deployment on Heroku

这是Flask Mega-Tutorial系列的第十八部分，我将在其中部署Microblog到Heroku云平台。

在前面的文章中，我向你展示了托管Python应用程序的“传统”方式，并且我演示了两个部署到Linux的服务器的实际示例。 如果你不曾管理过Linux系统，那么你可能认为需要投入大量工作到这项任务中，而且肯定会有一个更简单的方法。

在本章中，我将向你展示一种完全不同的部署方法，该方法依赖第三方云托管提供程序来执行大部分管理任务，从而使你能够腾出更多时间处理应用程序。

许多云托管提供商提供了一个应用程序可以运行的托管平台。 你只需提供部署到这些平台上的实际应用程序，因为硬件，操作系统，脚本语言解释器，数据库等都由该服务管理。 这种服务称为平台即服务（PaaS）。

是不是感到难以置信？

我将把Microblog部署到Heroku，这是一种流行的云托管服务，对Python应用程序也非常友好。 我选择Heroku不仅仅是因为它非常受欢迎，还因为它有一个免费的服务级别，可以让你跟随我并在不花钱的情况下完成部署。

本章的*GitHub*链接为：*Browse*, *Zip*, *Diff.*

托管于Heroku

Heroku是首批PaaS平台之一。 它以Ruby的应用程序的托管服务开始，随后逐渐发展到支持诸多其他语言，如Java，Node.js和Python。

在Heroku中部署Web应用程序主要是通过`git`版本控制工具完成的，因此你必须将应用程序放在git代码库中。 Heroku在应用程序的根目录中查找名为*Procfile*的文件，以获取有关如何启动应用程序的描述。 对于Python项目，Heroku还期望*requirements.txt*文件列出需要安装的所有模块依赖项。 在通过git将应用程序上传到Heroku的服务器之后，你的工作基本就完成了，只需等待几秒钟，应用程序就会上线。 整个操作流程就是这么简单。

Heroku提供不同的服务级别，允许你自主选择为应用程序提供多少计算能力和运行时间，随着用户群的增长，你需要购买更多的“dynos”计算单元。

准备好了吗？让我们开始吧！

创建Heroku账户

在部署应用到Heroku之前，你需要拥有一个帐户。 所以请访问heroku.com并创建一个免费账户。 一旦注册成功并登录到Heroku，你将可以访问一个dashboard，其中列出了你的所有应用程序。

安装Heroku命令行客户端

Heroku提供了一个名为Heroku CLI的命令行工具来与服务交互，可安装于Windows，Mac OS X和Linux。 该文档包括了支持的所有平台的安装说明。 如果你计划部署应用程序以测试该服务，请将其安装在你的系统上。 Heroku provides a command-line tool for interacting with their service called Heroku CLI, available for Windows, Mac OS X and Linux. The documentation includes installation instructions for all the supported platforms. Go ahead and install it on your system if you plan on deploying the application to test the service.

安装CLI后应该做的第一件事是登录到你的Heroku帐户：

```
$ heroku login
```

Heroku CLI会要求你输入电子邮件地址和帐户密码。 你的身份验证状态将在随后的命令中被记住。

设置Git

`git` 工具是Heroku应用程序部署的核心，因此如果你还没有安装它的话，则必须将它安装到你的系统上。 如果你没有可用于你的操作系统的安装包，可以访问git site下载安装程序。

使用 `git` 的原因很多并且都理由充分。 如果你打算部署应用到Heroku，那么这些原因就要又增加一个，因为要部署应用到Heroku，你的应用程序必须在 `git` 代码库中。 如果你要为Microblog执行测试部署，可以从GitHub克隆应用程序：

```
$ git clone https://github.com/miguelgrinberg/microblog
$ cd microblog
$ git checkout v0.18
```

`git checkout` 命令将代码库切换到指定的历史提交点，也就是本章所处的位置。

如果更喜欢使用你自己的代码，你可以通过在顶层目录中运行 `git init .` 来将你自己的项目转换成 `git` 代码库（注意 `init` 后面的句号，它告诉git你想要在当前目录中初始化代码库）。

创建Heroku应用

要用Heroku注册一个新应用，需要在应用程序根目录下使用 `apps:create` 子命令，并将应用程序名称作为唯一参数传递：

```
$ heroku apps:create flask-microblog
Creating flask-microblog... done
http://flask-microblog.herokuapp.com/ | https://git.heroku.com/flask-microblog.git
```

Heroku要求应用程序的名称具有唯一性。 我上面已使用了 `flask-microblog` 这个名称，所以你需要为你的部署选择一个不同的名称。

该命令的输出将包含Heroku分配给应用程序的URL以及git代码库。 你的本地git代码库将配置一个额外的*remote*，称为 `heroku` 。 你可以用 `git remote` 命令验证它是否存在：

```
$ git remote -v
heroku  https://git.heroku.com/flask-microblog.git (fetch)
heroku  https://git.heroku.com/flask-microblog.git (push)
```

根据你创建git代码库的方式，上述命令的输出还可能包含另一个名为 `origin` 的远程仓库地址。

临时文件系统

Heroku平台与其他部署平台不同之处在于它在虚拟化平台上运行的文件系统是临时的。 那是什么意思？ 这意味着Heroku可以随时将运行你的应用的虚拟服务器重置为干净状态。 你不该天真地认为你保存到文件系统的任何数据都会被持久存储，事实上，Heroku经常回收服务器。

在这种条件下工作会为我的应用程序带来一些问题，因为它使用了如下的几个文件：

- 默认的SQLite数据库引擎将数据写入磁盘文件中
- 应用程序的日志也写入磁盘文件中
- 编译的语言翻译存储库同样是本地文件

以下部分将针对这三个方面提出解决方案。

使用Heroku Postgres数据库

为了解决第一个问题，我将切换到不同的数据库引擎。 在第十七章中，你看到我使用MySQL数据库为Ubuntu部署添加健壮性。 Heroku基于Postgres数据库提供了自己的数据库产品，因此我将转而使用它来避免使用基于文件的SQLite。

Heroku应用的数据库使用相同的Heroku CLI进行设置。 在本章中，我将创建一个免费级别的数据库：

```
$ heroku addons:add heroku-postgresql:hobby-dev
Creating heroku-postgresql:hobby-dev on flask-microblog... free
Database has been created and is available
 ! This database is empty. If upgrading, you can transfer
 ! data from another database with pg:copy
Created postgresql-parallel-56076 as DATABASE_URL
Use heroku addons:docs heroku-postgresql to view documentation
```

新创建的数据库的URL存储在`DATABASE_URL`环境变量中，该变量在应用程序运行时将可用。这就非常方便了，因为应用程序已经设定为在该变量中查找数据库URL。

输出日志到标准输出

Heroku希望应用程序直接输出日志到`stdout`。 当你使用`heroku logs`命令时，应用程序打印到标准输出的任何内容都将被保存并返回。 所以我要添加一个配置变量，指示我是要输出日志到`stdout`，还是像我之前那样输出到文件。 这是配置的变化：

config.py：输出日志到标准输出的选项。

```
class Config(object):
    # ...
    LOG_TO_STDOUT = os.environ.get('LOG_TO_STDOUT')
```

然后在应用工厂函数中，我会检查此配置以了解应该如何配置应用程序的日志记录器：

`app/__init__.py`：输出日志到标准输出或文件。

```
def create_app(config_class=Config):
    # ...
    if not app.debug and not app.testing:
        # ...

        if app.config['LOG_TO_STDOUT']:
            stream_handler = logging.StreamHandler()
            stream_handler.setLevel(logging.INFO)
            app.logger.addHandler(stream_handler)
        else:
            if not os.path.exists('logs'):
                os.mkdir('logs')
            file_handler = RotatingFileHandler('logs/microblog.log',
                                               maxBytes=10240, backupCount=10)
            file_handler.setFormatter(logging.Formatter(
                '%(asctime)s %(levelname)s: %(message)s '
                '[in %(pathname)s:%(lineno)d]'))
            file_handler.setLevel(logging.INFO)
            app.logger.addHandler(file_handler)

        app.logger.setLevel(logging.INFO)
        app.logger.info('Microblog startup')

    return app
```

所以现在我需要在Heroku中运行应用程序时设置 `LOG_TO_STDOUT` 环境变量，但在其他配置中则不需要。 Heroku CLI使得做到这一点变得简单，因为它提供了一个选项来设置运行时使用的环境变量：

```
$ heroku config:set LOG_TO_STDOUT=1
Setting LOG_TO_STDOUT and restarting flask-microblog... done, v4
LOG_TO_STDOUT: 1
```

编译翻译

Microblog依赖本地文件的第三个方面是编译后的语言翻译文件。 确保这些文件永远不会从临时文件系统中消失的粗暴做法是将编译后的语言文件添加到git代码库，以便在部署到Heroku后它们成为应用程序初始状态的一部分。

在我看来，更优雅的选择是在Heroku的启动命令中包含 `flask translate compile` 命令，以便在服务器重新启动时再次编译这些文件。 我打算选择这个方案，因为我知道启动过程需要多个命令，至少我还需要运行数据库迁移。 所以现在，我将把这个问题放在一边，稍后当我写*Procfile*的时候会重新讨论它。

托管Elasticsearch

Elasticsearch是可以添加到Heroku项目中的众多服务之一，但与Postgres不同的是，这不是由Heroku提供的服务，而是由与Heroku合作提供附加组件的第三方提供的。 在我写这篇文章的时候，有三个不同的集成Elasticsearch服务提供商。

在配置Elasticsearch之前，请注意，Heroku要求你的帐户在安装任何第三方附加组件之前添加信用卡信息，即使你仍处于免费级别中。 如果你不想将信用卡信息提供给Heroku，请跳过此部分。 你仍然可以部署应用程序，但搜索功能不起作用。

在可作为附加组件提供的Elasticsearch选项中，我决定尝试SearchBox，它附带一个免费的初试计划。 要将SearchBox添加到你的帐户，你必须在登录到Heroku后运行以下命令：

```
$ heroku addons:create searchbox:starter
```

该命令将部署一个Elasticsearch服务，并将该服务的连接URL保存在与你的应用程序关联的 `SEARCHBOX_URL` 环境变量中。 请记住，除非将你的信用卡信息添加到你的Heroku帐户中，否则此命令将失败。

回忆一下第十六章，我的应用程序在Elasticsearch连接URL中查找的是 `ELASTICSEARCH_URL` 变量，所以我需要添加这个变量并将其设置为由SearchBox分配的连接URL：

```
$ heroku config:get SEARCHBOX_URL
<your-elasticsearch-url>
$ heroku config:set ELASTICSEARCH_URL=<your-elasticsearch-url>
```

在这里，我首先要求Heroku打印 `SEARCHBOX_URL` 的值，然后将其添加到一个名为 `ELASTICSEARCH_URL` 的新环境变量中。

更新依赖

Heroku期望依赖关系在*requirements.txt*文件中，就像我在第十五章中定义的那样。 但是为了在Heroku上运行应用程序，我需要为这个文件添加两个新的依赖关系。

Heroku不提供自己的Web服务器。 相反，它希望应用程序根据环境变量 `$PORT` 中给出的端口号启动自己的Web服务器。 由于Flask开发Web服务器不足以用于生产，因此我将再次使用gunicorn，这是Heroku为Python应用程序推荐的服务器。

该应用程序还将连接到Postgres数据库，为此SQLAlchemy依赖 `psycopg2` 软件包的安装。

`gunicorn` 和 `psycopg2` 都需要添加到*requirements.txt*文件中。

Procfile

Heroku需要知道如何执行应用程序，并且它会在应用程序的根目录中使用名为*Procfile*的文件。 这个文件的格式很简单，每行包含一个进程名称，一个冒号，然后是启动进程的命令。 在Heroku上运行的最常见的应用程序类型是一个Web应用程序，对于这种类型的应用程序，进程名称应该是 `web` 。 下面你可以看到Microblog的*Procfile*：

Procfile：Heroku Procfile。

```
web: flask db upgrade; flask translate compile; gunicorn microblog:app
```

在这里，我定义的启动命令中将按顺序执行三个命令作以启动Web应用程序。 首先，我运行数据库迁移升级，然后编译语言翻译，最后启动服务器。

因为前两个子命令是基于 `flask` 命令的，所以我需要添加 `FLASK_APP` 环境变量：

```
$ heroku config:set FLASK_APP=microblog.py
Setting FLASK_APP and restarting flask-microblog... done, v4
FLASK_APP: microblog.py
```

`gunicorn` 命令比我用于Ubuntu部署的还要简单，因为这个服务与Heroku环境有很好的集成。 例如， `$PORT` 环境变量默认会被设置，取代使用 `-w` 选项来设置worker的数量，heroku推荐添加一个名为 `WEB_CONCURRENCY` 的环境变量，在 `-w` 参数没有提供的时候，就会使用这个环境

变量，因此你可以灵活地控制worker的数量而无需修改Procfile。

部署应用

所有准备步骤都已完成，所以现在是时候执行部署了。 要将应用程序上传到Heroku的服务器进行部署，需要使用 `git push` 命令。 这与你将本地git代码库中的更改推送到GitHub或其他远程git服务器的方式类似。

现在我已经达到了最有趣的部分，就是将应用程序推送到我们的Heroku托管帐户。 这其实很简单，我只需要使用 `git` 将应用程序推送到Heroku git代码库的主分支就行了。 关于如何做到这一点有几种方法，取决于你是如何创建你的git代码库的。 如果你使用我的 `v0.18` 代码，那么你需要基于此标记创建一个分支，并将其作为远程主分支推送，如下所示：

```
$ git checkout -b deploy
$ git push heroku deploy:master
```

相反，如果你正在使用自己的代码库，那么你的代码已经在 `master` 分支中，所以你首先需要确保你的更改已经提交：

```
$ git commit -a -m "heroku deployment changes"
```

然后运行如下命令启动部署：

```
$ git push heroku master
```

无论你如何推送分支，都应该看到Heroku的以下输出：

```
$ git push heroku deploy:master
Counting objects: 247, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (238/238), done.
Writing objects: 100% (247/247), 53.26 KiB | 3.80 MiB/s, done.
Total 247 (delta 136), reused 3 (delta 0)
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Python app detected
remote: -----> Installing python-3.6.2
remote: -----> Installing pip
remote: -----> Installing requirements with pip
...
remote:
remote: -----> Discovering process types
remote:        Procfile declares types -> web
remote:
remote: -----> Compressing...
remote:        Done: 57M
remote: -----> Launching...
remote:        Released v5
remote:        https://flask-microblog.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/flask-microblog.git
 * [new branch]      deploy -> master
```

我们在`git push`命令中使用的标签`heroku`是在创建应用程序时由Heroku CLI自动添加的远程代码库。`deploy:master`参数意味着我将代码从本地代码库的`deploy`分支推送到Heroku代码库上的`master`分支。 当你使用自己的项目时，你可能会用`git push heroku master`命令推动你的本地`master`分支。 由于这个项目的代码库分支结构，我推送了一个非`master`的分支，但Heroku侧要求的目标分支是'master'，因为这是Heroku唯一接受部署的分支。

就这样，应用程序现在应该已经部署在创建应用程序的命令的输出中给出的URL上了。 在我的案例中，URL是*https://flask-microblog.herokuapp.com*，所以这就是我需要键入和访问该应用程序的URL。

如果你想查看正在运行的应用程序的日志，请使用`heroku logs`命令。 如果由于任何原因导致应用程序无法启动，该命令可能很有用。 如果有任何错误，将在日志中显示。

部署应用更新

要部署新版本的应用程序，只需要使用`git push`命令将新的代码库推送到Heroku即可。 这将重复部署过程，关停旧部署，然后用新代码替换它。 Procfile中的命令将作为新部署的一部分再次运行，因此在此过程中将更新任何新的数据库迁移或翻译内容。

本文翻译自The Flask Mega-Tutorial Part XIX: Deployment on Docker Containers

这是Flask Mega-Tutorial系列的第十九部分，我将在其中部署Microblog到Docker容器平台。

在第十七章中，你了解了传统部署，使用这种部署方式，你必须关注服务器配置的每个细节。 然后在第十八章我带你到另一个极端——Heroku ，这是一项完全掌控配置和部署任务的服务，使你能够全神贯注于应用程序。 在本章中，你将学习基于容器（尤其是在Docker容器平台）的第三种应用程序部署策略。 这种部署的工作量，介于另外两个选项之间。

容器建立在轻量级虚拟化技术的基础上，允许应用程序及其依赖和配置完全隔离宿主机地运行，而不需要使用虚拟机等完整的虚拟化解决方案。使用虚拟机需要更多的资源，并且有时可能与宿主机相比，性能显著下降。 配置为容器宿主机的系统可以运行大量容器，所有这些容器共享主机的内核并直接访问主机的硬件。 这与虚拟机不同，虚拟机必须模拟完整的系统，包括CPU，磁盘，其他硬件，内核等。

尽管必须共享内核，但容器中的隔离级别非常高。 容器具有自己的文件系统，并且可以基于容器宿主机使用不同的操作系统。 例如，你可以在Fedora宿主机上运行基于Ubuntu Linux的容器，反之亦然。 尽管容器是Linux操作系统上诞生的技术，但由于虚拟化的原因，也可以在Windows和Mac OS X宿主机上运行Linux容器。 这允许你在开发系统上测试部署操作，并且如果你愿意的话，还可以将容器合并到开发工作流程中去。

本章的*GitHub*链接为：*Browse*, *Zip*, *Diff.*

安装Docker社区版

尽管Docker不是唯一的容器平台，但它是迄今为止最受欢迎的，所以我选择了它。 有两个版本的Docker，免费的社区版（CE）和付费的企业版（EE）。 对于本教程来说，Docker CE就够了。

要使用Docker CE，首先必须将其安装在系统上。 在Docker网站上有适用于Windows，Mac OS X和多个Linux发行版的安装程序。 如果你正在使用Microsoft Windows系统，请务必注意Docker CE依赖Hyper-V。 如有必要，安装程序将为你启用此功能，但请记住，启用Hyper-V会限制诸如VirtualBox等其他虚拟化技术产品的运行。

一旦Docker CE安装在你的系统上，你可以通过在终端窗口或命令提示符处输入以下命令来验证安装是否成功：

```
$ docker version
Client:
 Version:      17.09.0-ce
 API version:  1.32
 Go version:   go1.8.3
 Git commit:   afdb6d4
 Built:        Tue Sep 26 22:40:09 2017
 OS/Arch:      darwin/amd64

Server:
 Version:      17.09.0-ce
 API version:  1.32 (minimum version 1.12)
 Go version:   go1.8.3
 Git commit:   afdb6d4
 Built:        Tue Sep 26 22:45:38 2017
 OS/Arch:      linux/amd64
 Experimental: true
```

构建容器镜像

为Microblog创建容器的第一步是为它构建一个镜像。 容器镜像是用于创建容器的模板。 它包含容器文件系统的完整表示，以及与网络，启动选项等相关的各种设置。

为应用程序创建容器镜像的最基本方法是启动一个要使用的基本操作系统（Ubuntu，Fedora等）容器，连接到运行在其中的bash shell进程，然后手动安装应用程序，可以参照我在第十七章中介绍的流程进行传统部署。 安装完所有内容后，你可以保存容器的快照，并生成容器镜像。 `docker` 命令支持这种类型的工作流，但我不打算讨论这种方法，因为它非常不便，每次需要生成新镜像时都必须手动安装应用程序。

更好的方法是通过脚本生成容器镜像。 创建脚本化容器镜像的命令是`docker build`。 该命令从一个名为*Dockerfile*的文件读取并执行构建指令（我需要创建这些指令）。 Dockerfile基本上可以认为是一个安装程序脚本，它执行安装步骤来部署应用程序，以及一些容器特定的设置。

这是Microblog的一份基础的*Dockerfile*：

Dockerfile: Microblog的Dockerfile。

```
FROM python:3.6-alpine

RUN adduser -D microblog

WORKDIR /home/microblog

COPY requirements.txt requirements.txt
RUN python -m venv venv
RUN venv/bin/pip install -r requirements.txt
RUN venv/bin/pip install gunicorn

COPY app app
COPY migrations migrations
COPY microblog.py config.py boot.sh ./
RUN chmod +x boot.sh

ENV FLASK_APP microblog.py

RUN chown -R microblog:microblog ./
USER microblog

EXPOSE 5000
ENTRYPOINT ["./boot.sh"]
```

Dockerfile中的每一行都是一条命令。 `FROM` 命令指定将在其上构建新镜像的基础容器镜像。这样一来，你从一个现有的镜像开始，添加或改变一些东西，并最终得到一个派生的镜像。镜像由名称和标签来标记，它们之间用冒号分隔。 该标签用作版本控制机制，允许容器镜像提供多个版本。 我选择的镜像的名称是 `python` ，它是Python的官方Docker镜像。 该镜像的标签允许你指定解释器版本和基础操作系统。 `3.6-alpine` 标签选择安装在Alpine Linux上的Python 3.6解释器。 由于其体积小，Alpine Linux发行版比起更常见的发行版（例如Ubuntu）会更多地被使用。 你可以在Python镜像库中查看Python镜像可用的标签。

`RUN` 命令在容器的上下文中执行任意命令。 这与你在shell提示符下输入命令相似。 `adduser -D microblog` 命令创建一个名为 `microblog` 的新用户。 大多数容器镜像都使用 `root` 作为默认用户，但以root身份运行应用程序并不是一个好习惯，所以我创建了自己的用户。

`WORKDIR` 命令设置将要安装应用程序的默认目录。 当我在上面创建 `microblog` 用户时，会自动创建了一个主目录，所以现在我将该目录设置为默认目录。 在Dockerfile中的任何剩余命令执行以及运行容器时，其当前目录为这个默认目录。

`COPY` 命令将文件从你的机器复制到容器文件系统。 该命令需要两个或更多参数，源文件/目录和目标文件/目录。 源文件必须与Dockerfile所在的目录相关。 目的地可以是绝对路径，也可以是相对于在之前的 `WORKDIR` 命令中设置的目录的路径。 在这第一个 `COPY` 命令中，我将*requirements.txt*文件复制到容器文件系统的 `microblog` 用户的主目录中。

容器中有了*requirements.txt*文件，我就可以使用 `RUN` 命令创建一个虚拟环境。 首先我创建它，然后在其中安装所有依赖。 由于依赖文件仅包含通用依赖项，因此我明确安装*gunicorn*，以将其用作Web服务器。 当然，我也可以在我的*requirements.txt*文件中添加gunicorn。

接下来的三个 `COPY` 命令从顶级目录中复制*app*包，含有数据库迁移的*migrations*目录以及中的*microblog.py*和*config.py*脚本。 我还复制了一个新文件，*boot.sh*，我将在下面讨论它。

`RUN chmod` 命令确保将这个新的*boot.sh*文件正确设置为可执行文件。 如果你使用的是基于Unix的文件系统，并且你的源文件已被标记为可执行文件，则复制的文件将会已是可执行的。 我显式地对其进行授权，是因为在Windows上很难设置可执行位。 如果你正在使用Mac OS X或Linux，你可能不需要这个步骤，但有了它也不会有什么问题。

`ENV` 命令在容器中设置环境变量。我需要设置 `FLASK_APP` ，它是 `flask` 命令所依赖的。

下面的 `RUN chown` 命令将存储在*/home/microblog*中的所有目录和文件的所有者设置为新的 `microblog` 用户。 尽管我在Dockerfile的顶部附近创建了该用户，但所有命令的默认用户仍为 `root` ，因此所有这些文件的属主都需要切换到 `microblog` 用户，以便在容器启动时该用户可以正确运行这些文件。

下一行中的 `USER` 命令使得这个新的 `microblog` 用户成为任何后续指令的默认用户，并且也是容器启动时的默认用户。

`EXPOSE` 命令配置该容器将用于服务的端口。 这是必要的，以便Docker可以适当地在容器中配置网络。 我选择了标准的Flask端口5000，但这其实可以是任意端口。

最后， `ENTRYPOINT` 命令定义了容器启动时应该执行的默认命令。 这是启动应用程序Web服务器的命令。 为了保持良好的代码组织逻辑，我决定为此创建一个单独的脚本，正是我之前复制到容器的*boot.sh*文件。 这里是这个脚本的内容：

boot.sh：Docker容器启动脚本。

```
#!/bin/sh
source venv/bin/activate
flask db upgrade
flask translate compile
exec gunicorn -b :5000 --access-logfile - --error-logfile - microblog:app
```

这是一个相当标准的启动脚本，与第十七章和第十八章的部署启动十分类似。 激活虚拟环境，执行迁移框架升级数据库，编译语言翻译，最后用gunicorn运行服务器。

请注意gunicorn命令之前的 `exec` 。 在shell脚本中， `exec` 触发正在运行脚本的进程被给定的命令来替换掉，而不是将这个命令作为新进程启动。 这很重要，因为Docker会将容器的生命与其上运行的第一个进程关联起来。 在像这样的情况下，启动进程不是容器的主进程，你需要确保主进程取代启动进程，以确保容器不会提前停止。

Docker的一个有趣的方面是容器写入 `stdout` 或 `stderr` 的任何内容都将被捕获并存储为容器的日志。 出于这个原因， `-access-logfile` 和 `--error-logfile` 都配置为 `-` ，它将日志发送到标准输出，以便它们作为日志由Docker存储。

Dockerfile写好后，我现在可以构建容器镜像了：

```
$ docker build -t microblog:latest .
```

我给`docker build`命令的`-t`参数设置了新容器镜像的名称和标签。`.`表示容器构建的基础目录，这就是*Dockerfile*所在的目录。构建过程将执行*Dockerfile*中的所有命令并创建镜像，该镜像将存储在你自己的机器上。

你可以使用`docker images`命令获取本地镜像的列表：

```
$ docker images
REPOSITORY    TAG          IMAGE ID       CREATED              SIZE
microblog     latest       54a47d0c27cf   About a minute ago   216MB
python        3.6-alpine   a6beab4fa70b   3 months ago         88.7MB
```

此列表将包含你的新镜像以及它的基础镜像。每当你对应用程序进行更改后，都可以通过再次运行build命令来更新容器镜像。

启动容器

使用已创建的镜像，你现在可以运行应用程序的容器版本。通过`docker run`命令，通常再搭配大量的参数，就可以完成容器的启动。我将首先向你展示一个基本的例子：

```
$ docker run --name microblog -d -p 8000:5000 --rm microblog:latest
021da2e1e0d390320248abf97dfbbe7b27c70fefed113d5a41bb67a68522e91c
```

`--name`选项为新容器提供了一个名称。`-d`选项告诉Docker在后台运行容器。如果没有`-d`，容器将作为前台应用程序运行，从而阻塞你的命令提示符。`-p`选项将容器端口映射到主机端口。第一个端口是主机上的端口，右边的端口是容器内的端口。上面的例子暴露了主机端口8000，其对应容器中的端口5000，因此即使内部容器使用5000，你也将在宿主机上访问端口8000来访问应用程序。一旦容器停止，`--rm`选项将使其自动被删除。虽然这不是必需的，但完成或中断的容器通常不再需要，因此可以自动删除。最后一个参数是容器使用的容器镜像名称和标签。运行上述命令后，可以在*http://localhost:8000*上访问该应用程序。

docker run的输出是分配给新容器的ID。这是一个很长的十六进制字符串，在随后的命令中你可以使用它来引用容器。实际上，只有前几个字符是必需的，足以保证ID的唯一性。

如果你想看看哪些容器正在运行，你可以使用`docker ps`命令：

```
$ docker ps
CONTAINER ID  IMAGE             COMMAND      PORTS                    NAMES
021da2e1e0d3  microblog:latest  "./boot.sh"  0.0.0.0:8000->5000/tcp   microblog
```

你可以看到，其实`docker ps`命令显示的是缩短了的容器ID。如果你现在想停止容器，你可以使用`docker stop`：

```
$ docker stop 021da2e1e0d3
021da2e1e0d3
```

回顾一下，应用程序配置中有许多来自环境变量的选项。 例如，Flask密钥，数据库URL和电子邮件服务器选项都是从环境变量中导入的。 在上面的 `docker run` 例子中，我没有考虑这些，因此所有这些配置选项都将使用默认值。

在更实际的例子中，你将在容器内设置这些环境变量。 你在前面的章节看到，*Dockerfile*中的ENV命令设置了环境变量，对于将变为静态的变量来说，这是一个方便的选项。 但是，对于依赖于安装的变量，将它们作为构建过程的一部分并不方便，因为你希望容器镜像具有良好的可移植性。 如果你想将应用程序作为容器镜像提供给另一个人，你希望该人员能够按原样使用它，而不必使用不同的变量重新构建它。

所以构建时的环境变量可能很有用，但是也需要有可以通过 `docker run` 命令设置的运行时环境变量，对于这些变量，可以使用 `-e` 选项来设置。 以下示例设置了密钥和gmail帐户：

```
$ docker run --name microblog -d -p 8000:5000 --rm -e SECRET_KEY=my-secret-key \
    -e MAIL_SERVER=smtp.googlemail.com -e MAIL_PORT=587 -e MAIL_USE_TLS=true \
    -e MAIL_USERNAME=<your-gmail-username> -e MAIL_PASSWORD=<your-gmail-password> \
    microblog:latest
```

由于具有许多环境变量定义， `docker run` 命令行非常长的情况并不罕见。

使用第三方“容器化”服务

Microblog的容器版本看起来不错，但我还没有真正考虑过很多关于存储的问题。 实际上，由于我没有设置 `DATABASE_URL` 环境变量，因此应用程序正在使用默认SQLite数据库并将数据存储在容器内部的文件系统上。 当你停止并删除容器时，你认为数据去哪里了？ 数据也会被删除！

容器中的文件系统是临时的，这意味着它随着容器的删除而删除。 你可以将数据写入容器内的文件系统，并且容器可以正常读写数据，但如果出于任何原因需要回收容器并将其替换为新的容器，则应用程序保存到容器内的任何数据将永远丢失。

容器应用程序的一个好的设计策略是保持应用程序容器无状态。 如果你的应用程序代码和数据容器没有任何问题，可以将其丢弃并替换为新的容器，容器变为真正的一次性容器，这在简化升级部署方面非常有用。

但是，这意味着数据必须放在应用程序容器之外的某个位置。 这就是神奇的Docker生态系统发挥作用的地方了。 Docker容器镜像仓库包含大量的容器镜像。你已经了解了Python容器镜像，我正在使用它作为我的Microblog容器的基础镜像。 除此之外，Docker还为Docker容器镜像仓库中的许多其他语言，数据库和其他服务维护镜像，如果这还不够，Docker容器镜像

仓库还允许公司为其产品发布容器镜像，并且像你我这样的常规用户也可以发布自己的镜像。 这意味着安装第三方服务需要做出的努力会减少成只需在Docker容器镜像仓库中找到合适的镜像，并通过带有适当参数的 `docker run` 命令启动它。

所以我现在要做的是创建两个额外的容器，一个用于MySQL数据库，另一个用于Elasticsearch服务，然后我将加长启动Microblog容器的命令， 以使其能够访问这两个新的容器。

添加MySQL容器

像许多其他产品和服务一样，MySQL在Docker镜像仓库中提供了公共容器镜像。 就像我自己的Microblog容器一样，MySQL依赖于需要传递给 `docker run` 的环境变量。 他们配置了密码，数据库名称等。在镜像仓库中有许多MySQL镜像时，我决定使用由MySQL官方团队维护的镜像。 你可以在其镜像仓库页面找到有关MySQL容器镜像的详细信息：*https://hub.docker.com/r/mysql/mysql-server/*。

回顾一下在第十七章中设置MySQL的繁琐过程，你就会赞叹在Docker中部署MySQL的轻松体验。 这里是启动MySQL服务器的 `docker run` 命令：

```
$ docker run --name mysql -d -e MYSQL_RANDOM_ROOT_PASSWORD=yes \
    -e MYSQL_DATABASE=microblog -e MYSQL_USER=microblog \
    -e MYSQL_PASSWORD=<database-password> \
    mysql/mysql-server:5.7
```

这就对了！ 在安装了Docker的任何机器上，你可以运行上面的命令，就会得到一个完成安装的MySQL服务器，它具有一个随机生成的root密码，一个名为 `microblog` 的全新数据库和一个名字相同的用户，该用户具备访问这个数据库的所有权限。 请注意，你需要输入正确的密码，以便它可以从 `MYSQL_PASSWORD` 环境变量获得。

现在在应用程序方面，我需要添加一个MySQL客户端软件包，就像我在Ubuntu上进行传统部署一样。 我将再次使用 `pymysql` ，我可以将它添加到*Dockerfile*中：

Dockerfile：添加pymysql到Dockerfile中。

```
# ...
RUN venv/bin/pip install gunicorn pymysql
# ...
```

任何时候对应用程序或*Dockerfile*进行更改后，都需要重建容器镜像：

```
$ docker build -t microblog:latest .
```

现在我可以再次启动Microblog，但是这次连接到数据库容器，以便两者都可以通过网络进行通信：

```
$ docker run --name microblog -d -p 8000:5000 --rm -e SECRET_KEY=my-secret-key \
    -e MAIL_SERVER=smtp.googlemail.com -e MAIL_PORT=587 -e MAIL_USE_TLS=true \
    -e MAIL_USERNAME=<your-gmail-username> -e MAIL_PASSWORD=<your-gmail-password> \
    --link mysql:dbserver \
    -e DATABASE_URL=mysql+pymysql://microblog:<database-password>@dbserver/microblog \
    microblog:latest
```

`--link` 选项告诉Docker让正要运行的容器可以访问参数中指定的容器。 该参数包含由冒号分隔的两个名称。 第一部分是要链接的容器的名称或ID，在本例中是我在上面创建的一个名为 `mysql` 的容器。 第二部分定义了一个可以在这个容器中用来引用链接的主机名。 这里我使用 `dbserver` 作为代表数据库服务器的通用名称。

通过建立两个容器之间的链接，我可以设置 `DATABASE_URL` 环境变量，以便SQLAlchemy被引导使用其他容器中的MySQL数据库。 数据库URL将使用 `dbserver` 作为数据库主机名， `microblog` 作为数据库名称和用户，以及你在启动MySQL时选择的密码。

我在试用MySQL容器时注意到的一件事是，这个容器需要几秒钟才能完全运行并准备好接受数据库连接。 如果启动MySQL容器，然后立刻启动应用容器，在*boot.sh*脚本尝试运行 `flask db migrate` 时，则可能会因数据库未准备好接受连接而失败。 为了使我的解决方案更加健壮，我决定在*boot.sh*中添加一个重试循环：

boot.sh：重试数据库连接。

```
#!/bin/sh
source venv/bin/activate
while true; do
    flask db upgrade
    if [[ "$?" == "0" ]]; then
        break
    fi
    echo Upgrade command failed, retrying in 5 secs...
    sleep 5
done
flask translate compile
exec gunicorn -b :5000 --access-logfile - --error-logfile - microblog:app
```

此循环检查 `flask db upgrade` 命令的退出代码，如果它不为零，则认为出现了问题，因此它会等待5秒钟然后重试。

添加Elasticsearch容器

Elasticsearch Docker文档演示了如何将该服务作为单一节点以用于开发模式，以及部署两个节点的生产环境服务。 现在，我将使用单节点模式，并使用引擎开源的“oss”镜像。 容器使用以下命令启动：

```
$ docker run --name elasticsearch -d -p 9200:9200 -p 9300:9300 --rm \
    -e "discovery.type=single-node" \
    docker.elastic.co/elasticsearch/elasticsearch-oss:6.1.1
```

这个 `docker run` 命令与我用于Microblog和MySQL的命令有很多相似之处，但是有一些有趣的区别。 首先，有两个 `-p` 选项，这意味着这个容器将在两个端口上而不是一个端口上进行监听。 端口9200和9300都映射到主机中的相同端口。

另一个区别在于用于引用容器镜像的语法。 对于我在本地构建的镜像，语法是 `<name>:<tag>` 。 MySQL容器使用格式为稍微更完整的 `<account>/<name>:<tag>` 语法，适用于在Docker镜像仓库中引用容器镜像。 我使用的Elasticsearch镜像遵循模式 `<registry>/<account><name>:<tag>` ，其中包括镜像仓库的地址作为第一个组件。 此语法用于未托管在Docker镜像仓库中的镜像。 在本处，Elasticsearch在*docker.elastic.co*上运行自己的容器镜像仓库服务，而不是使用由Docker维护的主镜像仓库。

所以，现在我已经启动并运行了Elasticsearch服务，我可以修改Microblog容器的启动命令以创建指向它的链接并设置Elasticsearch服务URL：

```
$ docker run --name microblog -d -p 8000:5000 --rm -e SECRET_KEY=my-secret-key \
    -e MAIL_SERVER=smtp.googlemail.com -e MAIL_PORT=587 -e MAIL_USE_TLS=true \
    -e MAIL_USERNAME=<your-gmail-username> -e MAIL_PASSWORD=<your-gmail-password> \
    --link mysql:dbserver \
    -e DATABASE_URL=mysql+pymysql://microblog:<database-password>@dbserver/microblog \
    --link elasticsearch:elasticsearch \
    -e ELASTICSEARCH_URL=http://elasticsearch:9200 \
    microblog:latest
```

在运行此命令之前，如果你仍然在运行Microblog容器，请先停止它。 还要仔细操作来为数据库设置正确的密码，并让Elasticsearch服务的参数处于命令中的恰当位置。

现在你应该可以访问*http://localhost:8000*并使用搜索功能。 如果你遇到任何错误，可以通过查看容器日志来对其进行排查。 你很可能希望查看Microblog容器的日志，其中将显示任何Python堆栈跟踪：

```
$ docker logs microblog
```

Docker容器镜像仓库

现在我已经在Docker上使用三个容器来运行了完整的应用程序，其中两个容器来自公开的第三方镜像。 如果你想提供自己的容器镜像给其他人，那么你必须将它们推送到任何人都可以获取到的Docker镜像仓库中。

要访问Docker镜像仓库，你需要转到*https://hub.docker.com*并为自己创建一个帐户。 确保你选择一个你喜欢的用户名，因为这将用于你发布的所有镜像。

为了能够从命令行访问你的账户，你需要使用 `docker login` 命令登录：

```
$ docker login
```

如果你一直跟随我的引导，现在你的计算机上已经有一个名为 `microblog:latest` 的镜像存储在本地。 为了能够将这个镜像推送到Docker镜像仓库中，它需要重新命名以包含该帐户，正如来自MySQL的镜像。 这是通过 `docker tag` 命令完成的：

```
$ docker tag microblog:latest <your-docker-registry-account>/microblog:latest
```

如果你再次用 `docker images` 列出你的镜像，你会看到两个Microblog条目，一个是 `microblog:latest` ，另一个还包括你的帐户名。 它们实际上是同一镜像的两个别名。

要将镜像发布到Docker镜像仓库，请使用 `docker push` 命令：

```
$ docker push <your-docker-registry-account>/microblog:latest
```

现在你的镜像被公开了，你可以像MySQL和服务那样，说明如何安装它并从Docker镜像仓库运行。

容器化应用的部署

让你的应用程序在Docker容器中运行的最大的好处之一是，一旦该容器在你的本地测试通过了，就可以将它们运行到任何提供Docker支持的平台。 例如，你可以使用第十七章中推荐的Digital Ocean，Linode或Amazon Lightsail上的相同服务器。 即使这些提供商提供的最便宜的产品也足以让Docker运行一些容器。

Amazon Container Service（ECS）使你能够创建一个容器宿主机集群，以在其中运行容器。在集成完备的AWS环境中，提供了水平扩展和负载平衡，以及为容器镜像使用私有容器镜像仓库的功能。

最后，容器编排平台例如Kubernetes通过允许你以简单的YAML格式文本文件描述你的多容器部署逻辑，来提供了更高级别的自动化和便利性， 负载均衡，水平扩展，密钥的安全管理以及滚动升级和回滚。

本文翻译自The Flask Mega-Tutorial Part XXI: User Notifications

这是Flask Mega-Tutorial系列的第二十一章，我将添加一个私有消息功能，它将在导航栏中显示用户通知，而且无需刷新页面就可以自动更新。

在本章中，我想继续致力于改进Microblog应用程序的用户体验。 有一个广泛应用的功能是向用户显示警报或通知。 社交应用通常会通过在顶部导航栏中显示带有数字的小徽章显示这些通知来让你知道有新的提及（@）或私有消息。 虽然这是最明显的用法，但通知模式还可以应用于许多其他类型的应用程序，以通知用户需要注意的事项。

为了向你展示构建用户通知所涉及的技术，我需要扩展Microblog。因此在本章的第一部分中，我将构建一个用户消息传递系统，它允许任何用户发送私有消息给另一个用户。 这实际上比听起来更简单，通过它，我们可以很好地复习核心的Flask实践，并告诉你Flask到底能在简单，高效和有趣的方面做到什么程度。 一旦消息系统就位，我就会讨论一些方法来实现显示未读消息计数的通知标志。

本章的*GitHub*链接为：*Browse*, *Zip*, *Diff.*

私有消息

我要实现的私有消息功能非常简单。 当你访问用户的个人主页时，会显示一个可以向该用户发送私有消息链接。 该链接将带你进入一个新的页面，在新页面中，可以在Web表单中发送消息。 要阅读发送给你的消息，页面顶部的导航栏将会有一个新的“消息”链接，它会将你带到与主页或发现页面相似的页面，但不会显示用户动态，它会显示其他用户发送给你的消息。

以下小节介绍了实现此功能所需的各个步骤。

私有消息的数据库支持

第一项任务是扩展数据库以支持私有消息。 这是一个新的 `Message` 模型：

app/models.py：Message模型。

```
class Message(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    sender_id = db.Column(db.Integer, db.ForeignKey('user.id'))
    recipient_id = db.Column(db.Integer, db.ForeignKey('user.id'))
    body = db.Column(db.String(140))
    timestamp = db.Column(db.DateTime, index=True, default=datetime.utcnow)

    def __repr__(self):
        return '<Message {}>'.format(self.body)
```

这个模型类与 `Post` 模型相似，唯一的区别是有两个用户外键，一个用于发信人，另一个用于收信人。 `User` 模型可以获得这两个用户的关系，以及一个新字段，用于指示用户最后一次阅读他们的私有消息的时间：

app/models.py：User模型对私有消息的支持。

```
class User(UserMixin, db.Model):
    # ...
    messages_sent = db.relationship('Message',
                                    foreign_keys='Message.sender_id',
                                    backref='author', lazy='dynamic')
    messages_received = db.relationship('Message',
                                        foreign_keys='Message.recipient_id',
                                        backref='recipient', lazy='dynamic')
    last_message_read_time = db.Column(db.DateTime)

    # ...

    def new_messages(self):
        last_read_time = self.last_message_read_time or datetime(1900, 1, 1)
        return Message.query.filter_by(recipient=self).filter(
            Message.timestamp > last_read_time).count()
```

这两个关系将返回给定用户发送和接收的消息，并且在关系的 `Message` 一侧将添加 `author` 和 `recipient` 回调引用。 我之所以使用 `author` 回调而不是更适合的 `sender` ，是因为通过使用 `author` ，我可以使用我用于用户动态的相同逻辑渲染这些消息。 `last_message_read_time` 字段将存储用户最后一次访问消息页面的时间，并将用于确定是否有比此字段更新时间戳的未读消息。 `new_messages()` 辅助方法实际上使用这个字段来返回用户有多少条未读消息。 在本章的最后，我将把这个数字作为页面顶部导航栏中的一个漂亮的徽章。

完成了数据库更改后，现在是时候生成新的迁移并使用它升级数据库了：

```
(venv) $ flask db migrate -m "private messages"
(venv) $ flask db upgrade
```

发送一条私有消息

下一步设计发送消息。我需要一个简单的Web表单来接收消息：

app/main/forms.py：私有消息表单类。

```
class MessageForm(FlaskForm):
    message = TextAreaField(_l('Message'), validators=[
        DataRequired(), Length(min=0, max=140)])
    submit = SubmitField(_l('Submit'))
```

而且我还需要在网页上呈现此表单的HTML模板：

app/templates/send_message.html：发送私有消息HTML模板。

```
{% extends "base.html" %}
{% import 'bootstrap/wtf.html' as wtf %}

{% block app_content %}
    <h1>{{ _('Send Message to %(recipient)s', recipient=recipient) }}</h1>
    <div class="row">
        <div class="col-md-4">
            {{ wtf.quick_form(form) }}
        </div>
    </div>
{% endblock %}
```

接下来，我将添加一个新的*/send_message/*路由来处理实际发送的私有消息：

app/main/routes.py：发送私有消息的视图函数。

```
from app.main.forms import MessageForm
from app.models import Message

# ...

@bp.route('/send_message/<recipient>', methods=['GET', 'POST'])
@login_required
def send_message(recipient):
    user = User.query.filter_by(username=recipient).first_or_404()
    form = MessageForm()
    if form.validate_on_submit():
        msg = Message(author=current_user, recipient=user,
                      body=form.message.data)
        db.session.add(msg)
        db.session.commit()
        flash(_('Your message has been sent.'))
        return redirect(url_for('main.user', username=recipient))
    return render_template('send_message.html', title=_('Send Message'),
                           form=form, recipient=recipient)
```

这个视图函数中的逻辑显而易见。 发送私有消息的操作只需在数据库中添加一个新的“消息”实例即可。

将所有内容联系在一起的最后一项更改是在用户个人主页中添加上述路由的链接：

app/templates/user.html：个人主页中添加发送私有消息的链接。

```
                {% if user != current_user %}
                <p>
                    <a href="{{ url_for('main.send_message',
                                        recipient=user.username) }}">
                        {{ _('Send private message') }}
                    </a>
                </p>
                {% endif %}
```

查看私有消息

这个功能的第二大部分是查看私有信息。 为此，我添加另一条路由*/messages*，该路由与主页和发现页面非常相似，包括分页的完全支持：

app/main/routes.py：查看消息视图函数。

```
@bp.route('/messages')
@login_required
def messages():
    current_user.last_message_read_time = datetime.utcnow()
    db.session.commit()
    page = request.args.get('page', 1, type=int)
    messages = current_user.messages_received.order_by(
        Message.timestamp.desc()).paginate(
            page, current_app.config['POSTS_PER_PAGE'], False)
    next_url = url_for('main.messages', page=messages.next_num) \
        if messages.has_next else None
    prev_url = url_for('main.messages', page=messages.prev_num) \
        if messages.has_prev else None
    return render_template('messages.html', messages=messages.items,
                           next_url=next_url, prev_url=prev_url)
```

我在这个视图函数中做的第一件事是用当前时间更新`User.last_message_read_time`字段。 这会将发送给该用户的所有消息标记为已读。 然后，我查询消息模型以获得消息列表，并按照最近的时间戳进行排序。我决定在这里复用`POSTS_PER_PAGE`配置项，因为用户动态和消息的页面看起来非常相似，但是如果发生了分歧，为消息添加单独的配置变量也是有意义的。 分页逻辑与我用于用户动态的逻辑完全相同，因此这对你来说应该很熟悉。

上面的视图函数通过渲染一个新的*/app/templates/messages.html*模板文件结束，该模板如下：

app/templates/messages.html：查看消息HTML模板。

```
{% extends "base.html" %}

{% block app_content %}
    <h1>{{ _('Messages') }}</h1>
    {% for post in messages %}
        {% include '_post.html' %}
    {% endfor %}
    <nav aria-label="...">
        <ul class="pager">
            <li class="previous{% if not prev_url %} disabled{% endif %}">
                <a href="{{ prev_url or '#' }}">
                    <span aria-hidden="true">&larr;</span> {{ _('Newer messages') }}
                </a>
            </li>
            <li class="next{% if not next_url %} disabled{% endif %}">
                <a href="{{ next_url or '#' }}">
                    {{ _('Older messages') }} <span aria-hidden="true">&rarr;</span>
                </a>
            </li>
        </ul>
    </nav>
{% endblock %}
```

在这里，我采取了另一个小技巧。 我注意到除了`Message`具有额外的`recipient`关系（我不需要在消息页面中显示，因为它总是当前用户），`Post`和`Message`实例具有几乎相同的结构。 所以我决定复用*app/templates/_post.html*子模板来渲染私有消息。 出于这个原因，这个模板使用了奇怪的for循环`for post in messages`，以便私有消息的渲染也可以套用到子模板上。

要让用户访问新的视图函数，导航页面需要生成一个新的“消息”链接：

app/templates/base.html：导航栏中的消息链接。

```
                {% if current_user.is_anonymous %}
                ...
                {% else %}
                <li>
                    <a href="{{ url_for('main.messages') }}">
                        {{ _('Messages') }}
                    </a>
                </li>
                ...
                {% endif %}
```

该功能现已完成，但作为所有更改的一部分，还有一些新的文本被添加到几个位置，并且需要将这些文本合并到语言翻译中。 第一步是更新所有的语言目录：

```
(venv) $ flask translate update
```

然后，*app/translations*中的每种语言都需要使用新翻译更新其*messages.po*文件。 你可以在本项目的GitHub代码库中找到西班牙语翻译，或者直接[下载zip文件]。

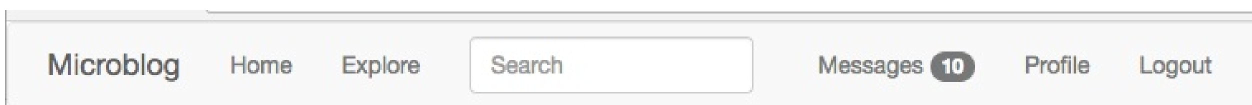
静态消息通知徽章

现在私有消息功能已经实现，但是还没有通过任何渠道告诉用户有私有消息等待阅读。导航栏上的未读消息标志的最简单实现可以使用Bootstrap badge小部件渲染到基础模板中：

app/templates/base.html：导航栏的静态消息通知徽章。

```
                ...
                <li>
                    <a href="{{ url_for('main.messages') }}">
                        {{ _('Messages') }}
                        {% set new_messages = current_user.new_messages() %}
                        {% if new_messages %}
                        <span class="badge">{{ new_messages }}</span>
                        {% endif %}
                    </a>
                </li>
                ...
```

在这里，我直接从模板中调用上面添加到User模型中的 `new_messages()` 方法，并将该数字存储在 `new_messages` 模板变量中。 然后，如果该变量不为零，我只需添加带有该数字的徽章到消息链接后面即可。 以下是这个页面的外观：



动态消息通知徽章

上一节介绍的解决方案是一种简单的常规方式来显示通知，但它有一个缺点，即徽章仅在加载新页面时刷新。 如果用户花费很长时间阅读一个页面上的内容而没有点击任何链接，那么在该时间内出现的新消息将不会显示，直到用户最终点击链接并加载新页面。

为了让这个应用程序对我的用户更有用，我希望徽章自行更新未读消息的数量，而用户不必点击链接并加载新页面。 上一节的解决方案的一个问题是，当加载页面时消息计数为非零时，徽章才在页面中渲染。 更方便的是始终在导航栏中包含徽章，并在消息计数为零时将其标记为隐藏。 这样可以很容易地使用JavaScript显示徽章：

app/templates/base.html：使用JavaScript渲染的友好未读消息徽章。

```
                    <li>
                        <a href="{{ url_for('main.messages') }}">
                            {{ _('Messages') }}
                            {% set new_messages = current_user.new_messages() %}
                            <span id="message_count" class="badge"
                                  style="visibility: {% if new_messages %}visible
                                                     {% else %}hidden {% endif %};">
                                {{ new_messages }}
                            </span>
                        </a>
                    </li>
```

使用此版本的徽章时，我总是将其包含在内，但当`new_messages`非零时，`visibility` CSS属性设置为`visible`；否则设置为`hidden`。 我还为表示徽章的元素添加了一个`id`属性，以便使用`$('#message_count')` jQuery选择器来简化这个元素的选取。

接下来，我编写一个简短的JavaScript函数，将该徽章更新为最新的数字：

app/templates/base.html：导航栏中的动态消息通知徽章

```
...
{% block scripts %}
    <script>
        // ...
        function set_message_count(n) {
            $('#message_count').text(n);
            $('#message_count').css('visibility', n ? 'visible' : 'hidden');
        }
    </script>
{% endblock %}
```

这个新的`set_message_count()`函数将设置徽章元素中的消息数量，并调整可见性，以便在计数为0时隐藏徽章。

向客户端发送通知

现在剩下的就是增加一种机制，通过这种机制，客户端可以定期接收有关用户拥有的未读消息数量的更新。 当更新发生时，客户端将调用 `set_message_count()` 函数来使用户知道更新。

实际上有两种方法可以让服务器将这些更新告知客户端，而且你可能会猜到，这两种方法都有优点和缺点，因此选择哪种方法很大程度上取决于项目。 在第一种方法中，客户端通过发送异步请求定期向服务器请求更新。 来自此请求的响应是更新列表，客户端可以使用这些更新来更新页面的不同元素，例如未读消息计数标记。 第二种方法需要客户端和服务器之间的特殊连接类型，以允许服务器自由地将数据推送到客户端。 请注意，无论采用哪种方法，我都希望将通知视为通用实体，以便我可以扩展此框架以支持除未读消息徽章以外的其他类型的事件。

第一种解决方案最大的优点是易于实施。 我需要做的只是向应用程序添加另一条路由，例如*/notifications*，它返回JSON格式的通知列表。然后客户端应用程序遍历通知列表并将必要的更改应用于页面。 该解决方案的缺点是实际事件和通知之间会有延迟，因为客户端会定期请求通知列表。 例如，如果客户端每10秒钟询问一次通知，则可能延迟10秒接收通知。

第二个解决方案需要在协议级别进行更改，因为HTTP没有服务器主动向客户端发送数据的任何规定。到目前为止，实现服务器推送消息的最常见方式是扩展服务器以支持除HTTP之外的WebSocket连接。 WebSocket是一种不同于HTTP的协议，在服务器和客户端之间建立永久连接。服务器和客户端可以随时向对方发送数据，而无需另一方请求。这种机制的优点是，无论何时发生客户感兴趣的事件，服务器都可以发送通知，而不会有任何延迟。缺点是WebSocket需要比HTTP更复杂的设置，因为服务器需要与每个客户端保持永久连接。想象一下，例如有四个worker进程的服务器通常可以服务几百个HTTP客户端，因为HTTP中的连接是短暂的并且不断被回收。而相同的服务器只能处理四个WebSocket客户端，在绝大多数情况下，这会导致资源紧张。正是由于这种限制，WebSocket应用程序通常围绕异步服务器进行设计，因为这种服务器在管理大量worker和活动连接方面效率更高。

好消息是，不管你使用什么方法，在客户端你都会有一个回调函数，它将被更新列表调用。因此，我可以从第一个解决方案开始，该解决方案实施起来要容易得多，如果发现不足，可以迁移到WebSocket服务器，该服务器可以配置为调用相同的客户端回调。 在我看来，对于这种类型的应用，第一种解决方案实际上是可以接受的。 基于WebSocket的实现对于需要以接近零延迟传递更新的应用程序非常有用。

这里有一些业界的类似案例。Twitter也使用的是第一种导航栏通知的方法；Facebook使用称为长轮询的HTTP变体，它解决了直接轮询的一些限制，同时仍然使用HTTP请求；Stack Overflow和Trello这两个站点使用WebSocket来实现通知机制。 你可以通过查看浏览器调试器的“Network”选项卡来查找任何网站上发生的后台活动请求。

我们继续实施轮询解决方案。 首先，我要添加一个新模型来跟踪所有用户的通知，以及用户模型中的关系。

app/models.py：通知模型。

```
import json
from time import time

# ...

class User(UserMixin, db.Model):
    # ...
    notifications = db.relationship('Notification', backref='user',
                                    lazy='dynamic')

    # ...

class Notification(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(128), index=True)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'))
    timestamp = db.Column(db.Float, index=True, default=time)
    payload_json = db.Column(db.Text)

    def get_data(self):
        return json.loads(str(self.payload_json))
```

通知将会有一个名称，一个关联的用户，一个Unix时间戳和一个有效载荷。 时间戳默认从 `time.time()` 函数中获取。 每种类型的通知都会有所不同，所以我将它写为JSON字符串，因为这样可以编写列表，字典或单个值（如数字或字符串）。 为了方便，我添加了 `get_data()` 方法，以便调用者不必操心JSON的反序列化。

这些更改需要包含在新的数据库迁移中：

```
(venv) $ flask db migrate -m "notifications"
(venv) $ flask db upgrade
```

为了方便，我将新增的 `Message` 和 `Notification` 模型添加到shell上下文，这样我就可以直接在用 `flask shell` 命令启动的解释器中使用这两个模型了。

microblog.py: 添加Message和Notification模型到shell上下文。

```
# ...
from app.models import User, Post, Notification, Message

# ...

@app.shell_context_processor
def make_shell_context():
    return {'db': db, 'User': User, 'Post': Post, 'Message': Message
            'Notification': Notification}
```

我还将在用户模型中添加一个 `add_notification()` 辅助方法，以便更轻松地处理这些对象：

app/models.py：Notification模型。

```
class User(UserMixin, db.Model):
    # ...

    def add_notification(self, name, data):
        self.notifications.filter_by(name=name).delete()
        n = Notification(name=name, payload_json=json.dumps(data), user=self)
        db.session.add(n)
        return n
```

此方法不仅为用户添加通知给数据库，还确保如果具有相同名称的通知已存在，则会首先删除该通知。 我将要使用的通知将被称为 `unread_message_count` 。 如果数据库已经有一个带有这个名称的通知，例如值为3，则当用户收到新消息并且消息计数变为4时，我就会替换旧的通知。

在任何未读消息数改变的地方，我需要调用 `add_notification()` ，以便我更新用户的通知，这样的地方有两处。 首先，在 `send_message()` 视图函数中，当用户收到一个新的私有消息时：

app/main/routes.py：更新用户通知。

```
@bp.route('/send_message/<recipient>', methods=['GET', 'POST'])
@login_required
def send_message(recipient):
    # ...
    if form.validate_on_submit():
        # ...
        user.add_notification('unread_message_count', user.new_messages())
        db.session.commit()
        # ...
    # ...
```

第二个地方是用户转到消息页面时，未读计数需要归零：

app/main/routes.py：查看消息视图函数。

```
@bp.route('/messages')
@login_required
def messages():
    current_user.last_message_read_time = datetime.utcnow()
    current_user.add_notification('unread_message_count', 0)
    db.session.commit()
    # ...
```

既然用户的所有通知都保存在数据库中，那么我可以添加一条新路由，客户端可以使用该路由为登录用户检索通知：

app/main/routes.py：通知视图函数。

```
from app.models import Notification

# ...

@bp.route('/notifications')
@login_required
def notifications():
    since = request.args.get('since', 0.0, type=float)
    notifications = current_user.notifications.filter(
        Notification.timestamp > since).order_by(Notification.timestamp.asc())
    return jsonify([{
        'name': n.name,
        'data': n.get_data(),
        'timestamp': n.timestamp
    } for n in notifications])
```

这是一个相当简单的函数，它返回一个包含用户通知列表的JSON负载。 每个通知都以包含三个元素的字典的形式给出，即通知名称，与通知有关的附加数据（如消息数量）和时间戳。 通知按照从创建时间顺序进行排序。

我不希望客户重复发送通知，所以我给他们提供了一个选项，只请求给定时间戳之后产生的通知。 `since` 选项可以作为浮点数包含在请求URL的查询字符串中，其中包含开始时间的unix时间戳。 如果包含此参数，则只有在此时间之后发生的通知才会被返回。

完成此功能的最后一部分是在客户端实现实际轮询。 最好的做法是在基础模板中实现，以便所有页面自动继承该行为：

app/templates/base.html：轮询通知。

```
...
{% block scripts %}
    <script>
        // ...
        {% if current_user.is_authenticated %}
        $(function() {
            var since = 0;
            setInterval(function() {
                $.ajax('{{ url_for('main.notifications') }}?since=' + since).done(
                    function(notifications) {
                        for (var i = 0; i < notifications.length; i++) {
                            if (notifications[i].name == 'unread_message_count')
                                set_message_count(notifications[i].data);
                            since = notifications[i].timestamp;
                        }
                    }
                );
            }, 10000);
        });
        {% endif %}
    </script>
```

该函数包含在一个模板条件中，因为我只想在用户登录时轮询新消息。对于没有登录的用户，这个函数将不会被渲染。

你已经在第二十章中看到了jQuery的 `$(function() { ...})` 模式。 这是注册一个函数在页面加载后执行的方式。 对于这个功能，我需要在页面加载时做的是设置一个定时器来获取用户的通知。 你还看到了 `setTimeout()` JavaScript函数，它在等待特定时间之后运行作为参数给

出的函数。 `setInterval()`函数使用与`setTimeout()`相同的参数，但不是一次性触发定时器，而是定期调用回调函数。 本处，我的间隔设置为10秒（以毫秒为单位），所以我将以每分钟大约六次的频率查看通知是否有更新。

利用定期计时器和Ajax，该函数轮询新通知路由，并在其完成回调中迭代通知列表。 当收到名为`unread_message_count`的通知时，通过调用上面定义的函数和通知中给出的计数来调整消息计数徽章。

我处理`since`参数的方式可能会令人困惑。 我首先将这个参数初始化为0。 参数总是包含在请求URL中，但是我不能像以前那样使用Flask的`url_for()`来生成查询字符串，因为一次请求中`url_for()`只在服务器上运行一次，而我需要`since`参数动态更新多次。 第一次，这个请求将被发送到*/notifications?since=0*，但是一旦我收到通知，我就会将`since`更新为它的时间戳。 这可以确保我不会收到重复的内容，因为我总是要求收到自我上次看到的通知以来发生的新通知。 同样重要的是要注意，我在interval函数外声明`since`变量，因为我不希望它是局部变量，我想要在所有调用中使用相同的变量。

最简单的测试方法是使用两种不同的浏览器A和B。 在两个浏览器上使用不同的用户登录Microblog。 然后从A浏览器向B浏览器上的用户发送一个或多个消息。 B浏览器的导航栏应更新为显示你在10秒钟内发送的消息数量。 而当你点击消息链接时，未读消息数重置为零。

本文翻译自The Flask Mega-Tutorial Part XXII: Background Jobs

这是Flask Mega-Tutorial系列的第二十二部分，我将告诉你如何创建独立于Web服务器之外运行的后台作业。

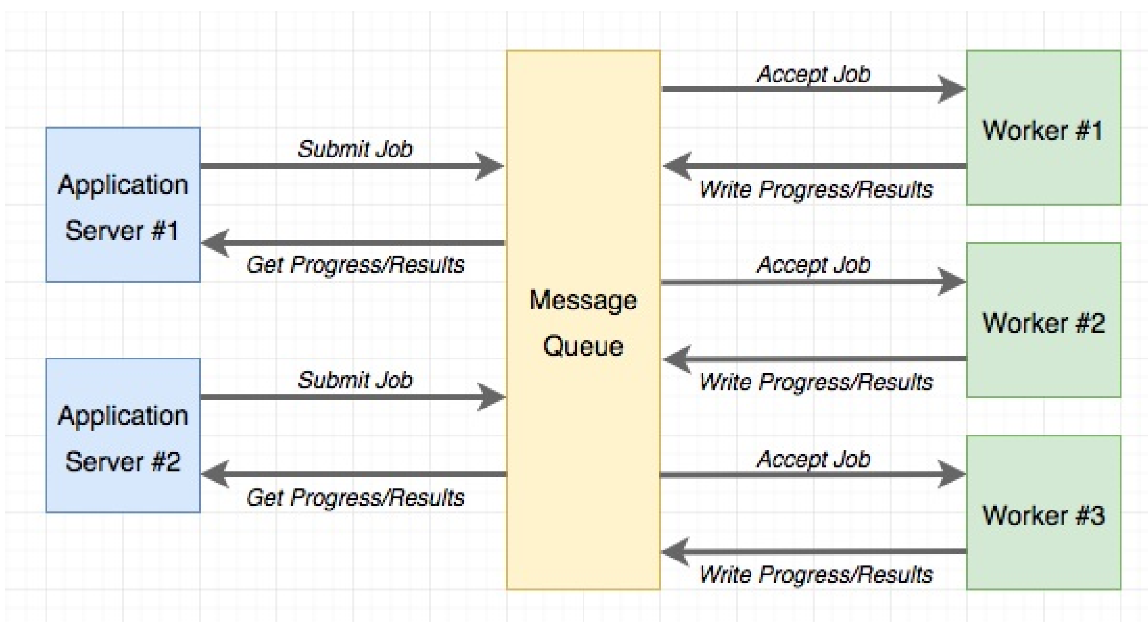
本章致力于为应用程序中运行时间较长或复杂的异步任务进程进行优化。这些进程不能在请求的上下文中同步执行，因为这会在任务持续期间阻塞对客户端的响应。在第十章中，我将邮件的发送转移到后台线程中执行，以免阻塞响应。 虽然使用线程处理电子邮件是可以接受的，但当问题处理时间更长时，此解决方案就不足以支撑了。 公认的做法是将耗时长的任务移交到worker进程（或进程池）。

为了证明长时间运行任务存在的必要性，我将介绍Microblog的一个导出功能，用户通过它可以请求一个包含他们所有用户动态的数据文件。 当用户使用该选项时，应用程序将启动一个导出任务，该导出任务将生成包含所有用户动态的JSON文件，然后通过电子邮件发送给用户。 所有这些活动都将在worker进程中发生，并且在执行时，用户可以看到显示完成百分比的进度。

本章的GitHub链接为：*Browse*, *Zip*, *Diff*.

任务队列简介

任务队列为后台作业提供了一个便捷的解决方案。 Worker进程独立于应用程序运行，甚至可以位于不同的系统上。 应用程序和worker之间的通信是通过消息队列完成的。 应用程序提交作业，然后通过与队列交互来监视其进度。 下图展示了一个典型的实现：



Python中最流行的任务队列是Celery。 这是一个相当复杂的软件包，它有很多选项并支持多个消息队列。 另一个流行的Python任务队列是Redis Queue(RQ)，它牺牲了一些灵活性，比如只支持Redis消息队列，但作为交换，它的建立要比Celery简单得多。

Celery和RQ都非常适合在Flask应用程序中支持后台任务，所以我倾向于选择更简单的RQ。不过，用Celery实现相同的功能其实也不难。如果你对Celery更感兴趣，可以阅读我的博客中的Using Celery with Flask文章。

使用RQ

RQ是一个标准的Python三方软件包，用 `pip` 安装：

```
(venv) $ pip install rq
(venv) $ pip freeze > requirements.txt
```

正如我前面提到的，应用和RQ worker之间的通信将在Redis消息队列中执行，因此你需要运行Redis服务器。有许多途径来安装和运行Redis服务器，比如下载其源码并执行编译和安装。如果你使用的是Windows，Microsoft在此处维护了Redis的安装程序。在Linux上，你可以通过操作系统的软件包管理器安装Redis。Mac OS X用户可以运行 `brew install redis`，然后使用 `redis-server` 命令手动启动服务。

除了确保服务正在运行并可供RQ访问之外，你不需要与Redis进行其他交互。

创建任务

通过RQ执行一项简单的任务后，你就会很快熟悉它。一个任务，不过是一个Python函数而已。以下是一个示例任务，我将其放入一个新的*app/tasks.py*模块：

app/tasks.py：示例后台任务。

```
import time

def example(seconds):
    print('Starting task')
    for i in range(seconds):
        print(i)
        time.sleep(1)
    print('Task completed')
```

该任务将秒数作为参数，然后在该时间量内等待，并每秒打印一次计数器。

运行RQ Worker

任务准备就绪，可以通过 `rq worker` 来启动一个worker进程了：

```
(venv) $ rq worker microblog-tasks
18:55:06 RQ worker 'rq:worker:miguelsmac.90369' started, version 0.9.1
18:55:06 Cleaning registries for queue: microblog-tasks
18:55:06
18:55:06 *** Listening on microblog-tasks...
```

Worker进程现在连接到了Redis，并在名为`microblog-tasks`的队列上查看可能分配给它的任何作业。 如果你想启动多个worker来扩展吞吐量，你只需要运行`rq worker`来生成更多连接到同一个队列的进程。 然后，当作业出现在队列中时，任何可用的worker进程都可以获取它。 在生产环境中，你可能希望至少运行可用CPU数量的worker。

执行任务

现在打开第二个终端窗口并激活虚拟环境。 我将使用shell会话来启动worker中的`example()`任务：

```
>>> from redis import Redis
>>> import rq
>>> queue = rq.Queue('microblog-tasks', connection=Redis.from_url('redis://'))
>>> job = queue.enqueue('app.tasks.example', 23)
>>> job.get_id()
'c651de7f-21a8-4068-afd5-8b982a6f6d32'
```

来自RQ的`Queue`类表示从应用程序端看到的任务队列。 它采用的参数是队列名称和一个`Redis`连接对象，本处使用默认URL进行初始化。 如果你的Redis服务器运行在不同的主机或端口号上，则需要使用其他URL。

Queue的`enqueue()`方法用于将作业添加到队列中。 第一个参数是要执行的任务的名称，可直接传入函数对象或导入字符串。 我发现传入字符串更加方便，因为不需要在应用程序的一端导入函数。 对`enqueue()`传入的任何剩余参数将被传递给worker中运行的函数。

只要进行了`enqueue()`调用，运行着RQ worker的终端窗口上就会出现一些活动。 你会看到`example()`函数正在运行，并且每秒打印一次计数器。 同时，你的其他终端不会被阻塞，你可以继续在shell中执行表达式。在上面的例子中，我调用`job.get_id()`方法来获取分配给任务的唯一标识符。 你可以尝试使用另一个有趣表达式来检查worker上的函数是否已完成：

```
>>> job.is_finished
False
```

如果你像我在上面的例子中那样传递了`23`，那么函数将运行约23秒。 在那之后，`job.is_finished`表达式将变为`True`。 就是这么简单，炫酷否？

一旦函数完成，worker又回到等待作业的状态，所以如果你想进行更多的实验，你可以用不同的参数重复执行`enqueue()`调用。 队列中存储的有关任务的数据将保留一段时间（默认为500秒），但最终会被删除。 这很重要，任务队列不保留已执行作业的历史记录。

报告任务进度

上面使用的示例任务简单得不现实。 通常，对于长时间运行的任务，你需要将一些进度信息提供给应用程序，从而可以将其显示给用户。 RQ通过使用作业对象的`meta`属性来支持这一点。 让我重写`example()`任务来编写进度报告：

app/tasks.py:：带进度的示例后台任务。

```
import time
from rq import get_current_job

def example(seconds):
    job = get_current_job()
    print('Starting task')
    for i in range(seconds):
        job.meta['progress'] = 100.0 * i / seconds
        job.save_meta()
        print(i)
        time.sleep(1)
    job.meta['progress'] = 100
    job.save_meta()
    print('Task completed')
```

这个新版本的`example()`使用RQ的`get_current_job()`函数来获取一个作业实例，该实例与提交任务时返回给应用程序的实例类似。 作业对象的`meta`属性是一个字典，任务可以编写任何想要与应用程序通信的自定义数据。 在这个例子中，我写入了`progress`，表示完成任务的百分比。 每次进程更新时，我都调用`job.save_meta()`指示RQ将数据写入Redis，应用程序可以在其中找到它。

在应用程序方面（目前只是一个Python shell），我可以运行此任务，然后监视进度，如下所示：

```
>>> job = queue.enqueue('app.tasks.example', 23)
>>> job.meta
{}
>>> job.refresh()
>>> job.meta
{'progress': 13.043478260869565}
>>> job.refresh()
>>> job.meta
{'progress': 69.56521739130434}
>>> job.refresh()
>>> job.meta
{'progress': 100}
>>> job.is_finished
True
```

如你所见，在另一侧，`meta`属性可以被读取。 需要调用`refresh()`方法来从Redis更新内容。

任务的数据库表示

对于上面的例子来说，启动一个任务并观察它运行就足够了。 对于Web应用程序，情况会变得更复杂一些，因为一旦任务随着请求的处理而启动，该请求随即结束，而该任务的所有上下文都将丢失。 因为我希望应用程序跟踪每个用户正在运行的任务，所以我需要使用数据库表来维护状态。 你可以在下面看到新的`Task`模型实现：

app/models.py：Task模型。

```
# ...
import redis
import rq

class User(UserMixin, db.Model):
    # ...
    tasks = db.relationship('Task', backref='user', lazy='dynamic')

# ...

class Task(db.Model):
    id = db.Column(db.String(36), primary_key=True)
    name = db.Column(db.String(128), index=True)
    description = db.Column(db.String(128))
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'))
    complete = db.Column(db.Boolean, default=False)

    def get_rq_job(self):
        try:
            rq_job = rq.job.Job.fetch(self.id, connection=current_app.redis)
        except (redis.exceptions.RedisError, rq.exceptions.NoSuchJobError):
            return None
        return rq_job

    def get_progress(self):
        job = self.get_rq_job()
        return job.meta.get('progress', 0) if job is not None else 100
```

这个模型和以前的模型有一个有趣的区别是 `id` 主键字段是字符串类型，而不是整数类型。这是因为对于这个模型，我不会依赖数据库自己的主键生成，而是使用由RQ生成的作业标识符。

该模型将存储符合任务命名规范的名称（会传递给RQ），适用于向用户显示的任务描述，该任务的所属用户的关系以及任务是否已完成的布尔值。 `complete` 字段的目的是将正在运行的任务与已完成的任务分开，因为运行中的任务需要特殊处理才能显示最新进度。

`get_rq_job()` 辅助方法可以用给定的任务ID加载RQ `Job` 实例。 这是通过 `Job.fetch()` 完成的，它会从Redis中存在的数据中加载 `Job` 实例。 `get_progress()` 方法建立在 `get_rq_job()` 的基础之上，并返回任务的进度百分比。 该方法做一些有趣的假设，如果模型中的作业ID不存在于RQ队列中，则表示作业已完成并且数据已过期并已从队列中删除，因此在这种情况下返回的百分比为100。 另一方面，如果job存在，但'meta'属性中找不到进度相关的信息，那么可以安全地假定该job计划运行，但还没有启动，所以在这种情况下进度是0。

要将更改应用于数据库，需要生成新的迁移，然后升级数据库：

```
(venv) $ flask db migrate -m "tasks"
(venv) $ flask db upgrade
```

新模型也可以添加到shell上下文中，以便在shell会话中访问它时无需导入：

microblog.py：添加Task模型到shell上下文中。

```
from app import create_app, db, cli
from app.models import User, Post, Message, Notification, Task

app = create_app()
cli.register(app)

@app.shell_context_processor
def make_shell_context():
    return {'db': db, 'User': User, 'Post': Post, 'Message': Message,
            'Notification': Notification, 'Task': Task}
```

将RQ与Flask应用集成

Redis服务的连接URL需要添加到配置中：

```
class Config(object):
    # ...
    REDIS_URL = os.environ.get('REDIS_URL') or 'redis://'
```

与往常一样，Redis连接URL将来自环境变量，如果该变量未定义，则会假定该服务在当前主机的默认端口上运行并使用默认URL。

应用工厂函数将负责初始化Redis和RQ：

*app/**init**.py*：整合RQ。

```
# ...
from redis import Redis
import rq

# ...

def create_app(config_class=Config):
    # ...
    app.redis = Redis.from_url(app.config['REDIS_URL'])
    app.task_queue = rq.Queue('microblog-tasks', connection=app.redis)

    # ...
```

`app.task_queue` 将成为提交任务的队列。 将队列附加到应用上会提供很大的便利，因为我可以在应用的任何地方使用 `current_app.task_queue` 来访问它。 为了方便应用的任何部分提交或检查任务，我可以在 `User` 模型中创建一些辅助方法：

app/models.py：用户模型中的任务辅助方法。

```
# ...

class User(UserMixin, db.Model):
    # ...

    def launch_task(self, name, description, *args, **kwargs):
        rq_job = current_app.task_queue.enqueue('app.tasks.' + name, self.id,
                                                *args, **kwargs)
        task = Task(id=rq_job.get_id(), name=name, description=description,
                    user=self)
        db.session.add(task)
        return task

    def get_tasks_in_progress(self):
        return Task.query.filter_by(user=self, complete=False).all()

    def get_task_in_progress(self, name):
        return Task.query.filter_by(name=name, user=self,
                                    complete=False).first()
```

`launch_task()`方法负责将任务提交到RQ队列，并将其添加到数据库中。 `name`参数是函数名称，如*app/tasks.py*中所定义的那样。 提交给RQ时，该函数会将`app.tasks.`预先添加到该名称中以构建符合规范的函数名称。 `description`参数是对呈现给用户的任务的友好描述。 对于导出用户动态的函数，我将名称设置为`export_posts`，将描述设置为`Exporting posts...`。 其余参数将传递给任务函数。 `launch_task()`函数首先调用队列的`enqueue()`方法来提交作业。 返回的作业对象包含由RQ分配的任务ID，因此我可以使用它在我的数据库中创建相应的`Task`对象。

请注意，`launch_task()`将新的任务对象添加到会话中，但不会发出提交。 一般来说，最好在更高层次函数中的数据库会话上进行操作，因为它允许你在单个事务中组合由较低级别函数所做的多个更新。 这不是一个严格的规则，并且，在本章后面的子函数中也会存在一个例外的提交。

`get_tasks_in_progress()`方法返回该用户未完成任务的列表。 稍后你会看到，我使用此方法在将有关正在运行的任务的信息渲染到用户的页面中。

最后，`get_task_in_progress()`是上一个方法的简化版本并返回指定的任务。 我阻止用户同时启动两个或多个相同类型的任务，因此在启动任务之前，可以使用此方法来确定前一个任务是否还在运行。

利用RQ任务发送电子邮件

不要认为本节偏离主题，我在上面说过，当后台导出任务完成时，将使用包含所有用户动态的JSON文件向用户发送电子邮件。 我在第十章中构建的电子邮件功能需要通过两种方式进行扩展。 首先，我需要添加对文件附件的支持，以便我可以附加JSON文件。 其次，`send_email()`函数总是使用后台线程异步发送电子邮件。 当我要从后台任务发送一封电子邮件时（已经是异步的了），基于线程的二级后台任务没有什么意义，所以我需要同时支持同步和异步电子邮件的发送。

幸运的是，Flask-Mail支持附件，所以我需要做的就是扩展`send_email()`函数的默认关键字参数，然后在`Message`对象中配置它们。 选择在前台发送电子邮件时，我只需要添加一个`sync=True`的关键字参数即可：

app/email.py：发送带附件的邮件。

```
# ...

def send_email(subject, sender, recipients, text_body, html_body,
               attachments=None, sync=False):
    msg = Message(subject, sender=sender, recipients=recipients)
    msg.body = text_body
    msg.html = html_body
    if attachments:
        for attachment in attachments:
            msg.attach(*attachment)
    if sync:
        mail.send(msg)
    else:
        Thread(target=send_async_email,
            args=(current_app._get_current_object(), msg)).start()
```

Message类的`attach()`方法接受三个定义附件的参数：文件名，媒体类型和实际文件数据。 文件名就是收件人看到的与附件关联的名称。 媒体类型定义了这种附件的类型，这有助于电子邮件读者适当地渲染它。 例如，如果你发送`image/png`作为媒体类型，则电子邮件阅读器会知道该附件是一个图像，在这种情况下，它可以显示它。 对于用户动态数据文件，我将使用JSON格式，该格式使用`application/json`媒体类型。 最后一个参数包含附件内容的字符串或字节序列。

简单来说，`send_email()`的`attachments`参数将成为一个元组列表，每个元组将有三个元素对应于`attach()`的三个参数。 因此，我需要将此列表中的每个元素作为参数发送给`attach()`。 在Python中，如果你想将列表或元组中的每个元素作为参数传递给函数，你可以使用`func(*args)`将这个列表或元祖解包成函数中的多个参数，而不必枯燥地一个个地传递，如`func(args[0], args[1], args[2])`。 例如，如果你有一个列表`args = [1, 'foo']`，`func(*args)`将会传递两个参数，就和你调用`func(1, 'foo')`一样。 如果没有`*`，调用将会传入一个参数，即`args`列表。

至于电子邮件的同步发送，我需要做的就是，当`sync`是`True`的时候恢复成调用`mail.send(msg)`。

任务助手

尽管我上面使用的`example()`任务是一个简单的独立函数，但导出用户动态的函数却需要应用中具有的一些功能，例如访问数据库和发送电子邮件。 因为这将在单独的进程中运行，所以我需要初始化Flask-SQLAlchemy和Flask-Mail，而Flask-Mail又需要Flask应用实例以从中获取它们的配置。 因此，我将在*app/tasks.py*模块的顶部添加Flask应用实例和应用上下文：

app/tasks.py：创建应用及其上下文。

```
from app import create_app

app = create_app()
app.app_context().push()
```

应用在此模块中创建，因为这是RQ worker要导入的唯一模块。 当使用 `flask` 命令时，根目录中的*microblog.py*模块创建应用实例，但RQ worker对此却一无所知，所以当任务函数需要它时，它需要创建自己的应用实例。 你已经在好几个地方看到了 `app.app_context()` 方法，推送一个上下文使应用成为“当前”的应用实例，这样一来Flask-SQLAlchemy等插件才可以使用 `current_app.config` 获取它们的配置。 没有上下文， `current_app` 表达式会返回一个错误。

然后我开始考虑如何在这个函数运行时报告进度。除了通过 `job.meta` 字典传递进度信息之外，我还想将通知推送给客户端，以便自动动态更新完成百分比。为此，我将使用我在第二十一章中构建的通知机制。更新将以与未读消息徽章非常类似的方式工作。当服务器渲染模板时，它将包含从 `job.meta` 获得的“静态”进度信息，但是一旦页面位于客户端的浏览器中，通知将使用通知来动态更新百分比。由于通知的原因，更新正在运行的任务的进度将比上一个示例中的操作稍微多一些，所以我将创建一个专用于更新进度的包装函数：

app/tasks.py：设置任务进度。

```
from rq import get_current_job
from app import db
from app.models import Task

# ...

def _set_task_progress(progress):
    job = get_current_job()
    if job:
        job.meta['progress'] = progress
        job.save_meta()
        task = Task.query.get(job.get_id())
        task.user.add_notification('task_progress', {'task_id': job.get_id(),
                                                     'progress': progress})
        if progress >= 100:
            task.complete = True
        db.session.commit()
```

导出任务可以调用 `_set_task_progress()` 来记录进度百分比。 该函数首先将百分比写入 `job.meta` 字典并将其保存到Redis，然后从数据库加载相应的任务对象，并使用 `task.user` 已有的 `add_notification()` 方法将通知推送给请求该任务的用户。 通知将被命名为 `task_progress` ，并且与其关联的数据将成为具有两个条目的字典：任务标识符和进度数值。 稍后我将添加JavaScript代码来处理这种新的通知类型。

该函数查看进度来确认任务函数是否已完成，并在这种情况下更新数据库中任务对象的 `complete` 属性。 数据库提交调用确保通过 `add_notification()` 添加的任务和通知对象都立即保存到数据库。 我需要非常精确地设计父任务，确保不执行任何数据库更改，因为执行本调用会将父任务的更改也写入数据库。

实现导出任务

现在所有的准备工作已经完成，可以开始编写导出函数了。 这个函数的高层结构如下：

app/tasks.py：导出用户动态通用结构。

```
def export_posts(user_id):
    try:
        # read user posts from database
        # send email with data to user
    except:
        # handle unexpected errors
```

为什么将整个任务包装在try/except块中呢？ 请求处理器中的应用代码可以防止意外错误，因为Flask本身捕获异常，然后将它们以我设置的日志配置的方式来进行处理。 然而，这个函数将运行在由RQ控制的单独进程中，而非Flask，因此如果发生任何意外错误，任务将中止，RQ将向控制台显示错误，然后返回等待新的job。 所以基本上，除非你正在观看RQ worker的输出或将其记录到文件中，否则将永远不会发现有错误。

让我们从上面带有注释的三部分中最简单的错误处理部分开始梳理：

app/tasks.py：导出用户动态错误处理。

```
import sys
# ...

def export_posts(user_id):
    try:
        # ...
    except:
        _set_task_progress(100)
        app.logger.error('Unhandled exception', exc_info=sys.exc_info())
```

每当发生意外错误时，我将通过将进度设置为100%来将任务标记为完成，然后使用Flask应用程序中的日志记录器对象记录错误以及堆栈跟踪信息（调用 `sys.exc_info()` 来获得）。 使用Flask应用日志记录器来记录错误的好处在于，你可以观察到你为Flask应用实现的任何日志记录机制。 例如，在第七章中，我配置了要发送到管理员电子邮件地址的错误。 只要使用 `app.logger` ，我也可以得到这些错误信息。

接下来，我将编写实际的导出代码，它只需发出一个数据库查询并在循环中遍历结果，并将它们累积在字典中：

app/tasks.py：从数据库读取用户动态。

```
import time
from app.models import User, Post

# ...

def export_posts(user_id):
    try:
        user = User.query.get(user_id)
        _set_task_progress(0)
        data = []
        i = 0
        total_posts = user.posts.count()
        for post in user.posts.order_by(Post.timestamp.asc()):
            data.append({'body': post.body,
                         'timestamp': post.timestamp.isoformat() + 'Z'})
            time.sleep(5)
            i += 1
            _set_task_progress(100 * i // total_posts)

        # send email with data to user
    except:
        # ...
```

每条动态都是一个包含两个条目的字典，即动态正文和动态发表的时间。 时间格式将采用ISO 8601标准。 我使用的Python的 `datetime` 对象不存储时区，因此在以ISO格式导出时间后，我添加了'Z'，它表示UTC。

由于需要跟踪进度，代码变得稍微复杂了些。 我维护了一个计数器 `i` ，并且在进入循环之前还需要发出一个额外的数据库查询，查询 `total_posts` 以获得用户动态的总数。 使用了 `i` 和 `total_posts` ，在每个循环迭代我都可以使用从0到100的数字来更新任务进度。

你可能会好奇我为什么会在每个循环迭代中加入 `time.sleep(5)` 调用。主要原因是我想要延长导出所需的时间，以便在用户动态不多的情况下也可以方便地查看到导出进度的增长。

下面是函数的最后部分，将会带上 `data` 附件发送邮件给用户：

app/tasks.py：发送带用户动态的邮件给用户。

```
import json
from flask import render_template
from app.email import send_email

# ...

def export_posts(user_id):
    try:
        # ...

        send_email('[Microblog] Your blog posts',
                sender=app.config['ADMINS'][0], recipients=[user.email],
                text_body=render_template('email/export_posts.txt', user=user),
                html_body=render_template('email/export_posts.html', user=user),
                attachments=[('posts.json', 'application/json',
                              json.dumps({'posts': data}, indent=4))],
                sync=True)
    except:
        # ...
```

其实只是对 `send_email()` 函数的调用。 附件被定义为一个元组，其中有三个元素被传递给Flask-Mail的 `Message` 对象的 `attach()` 方法。 元组中的第三个元素是附件内容，它是用Python的 `json.dumps()` 函数生成的。

这里引用了一对新模板，它们以纯文本和HTML格式提供电子邮件正文的内容。 这是文本模板的内容：

app/templates/email/export_posts.txt：导出用户动态文本邮件模板。

```
Dear {{ user.username }},

Please find attached the archive of your posts that you requested.

Sincerely,

The Microblog Team
```

这是HTML版本的邮件模板：

app/templates/email/export_posts.html：导出用户动态HTML邮件模板。

```
<p>Dear {{ user.username }},</p>
<p>Please find attached the archive of your posts that you requested.</p>
<p>Sincerely,</p>
<p>The Microblog Team</p>
```

应用中的导出功能

所有支持后台导出任务的核心组件现已到位。 剩下的就是将这个功能连接到应用，以便用户发起请求并通过电子邮件发送用户动态给他们。

下面是新的 `export_posts` 视图函数：

app/main/routes.py：导出用户动态路由和视图函数。

```
@bp.route('/export_posts')
@login_required
def export_posts():
    if current_user.get_task_in_progress('export_posts'):
        flash(_('An export task is currently in progress'))
    else:
        current_user.launch_task('export_posts', _('Exporting posts...'))
        db.session.commit()
    return redirect(url_for('main.user', username=current_user.username))
```

该函数首先检查用户是否有未完成的导出任务，并在这种情况下只是闪现消息。 对同一用户同时执行两个导出任务是没有意义的，可以避免。 我可以使用前面实现的 `get_task_in_progress()` 方法来检查这种情况。

如果用户没有正在运行的导出任务，则调用`launch_task()`来启动它。 第一个参数是将传递给RQ worker的函数的名称，前缀为`app.tasks.`。 第二个参数只是一个友好的文本描述，将会显示给用户。 这两个值都会被写入数据库中的Task对象。 该函数以重定向到用户个人主页结束。

现在我需要暴露该路由的链接，以便用户可以请求导出。 我认为最合适的地方是在用户个人主页，只有在用户查看他们自己的主页时，链接在“编辑个人资料”链接下面显示：

app/templates/user.html：用户个人主页的导出链接。

```
                ...
                <p>
                    <a href="{{ url_for('main.edit_profile') }}">
                        {{ _('Edit your profile') }}
                    </a>
                </p>
                {% if not current_user.get_task_in_progress('export_posts') %}
                <p>
                    <a href="{{ url_for('main.export_posts') }}">
                        {{ _('Export your posts') }}
                    </a>
                </p>
                ...
                {% endif %}
```

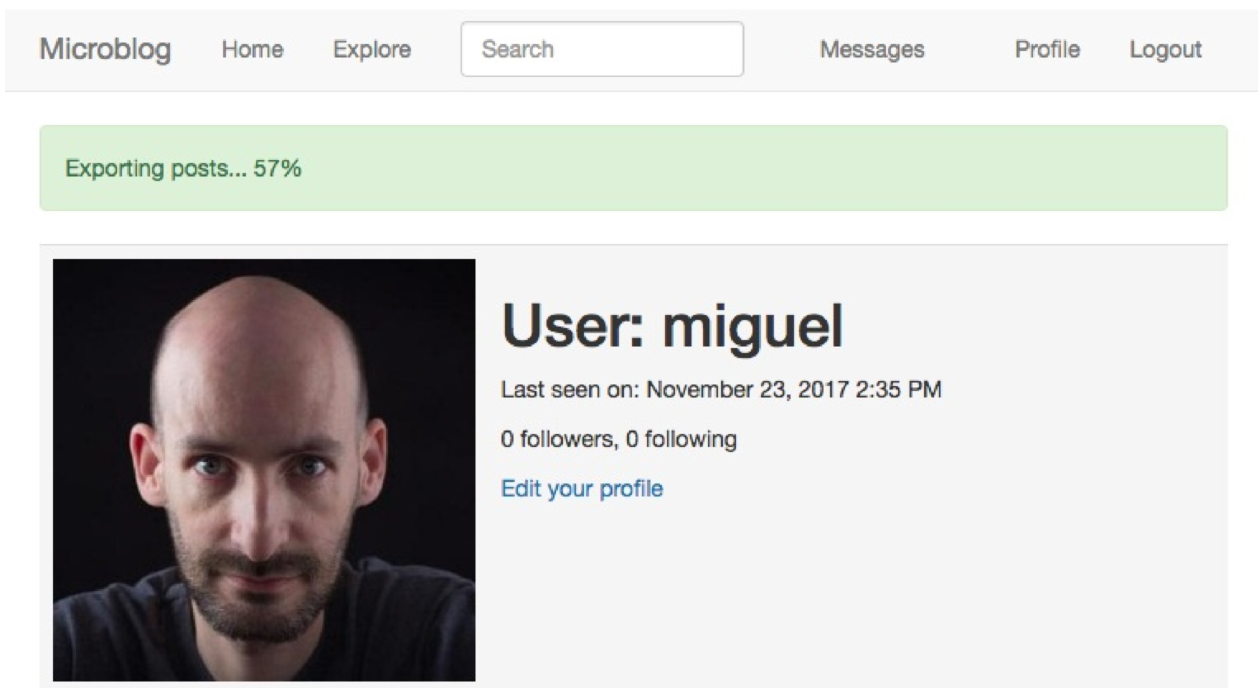
此链接的渲染是有条件的，因为我不希望它在用户已经有导出任务执行时出现。

此时的后台作业是可以运作的，但是不会向用户提供任何反馈。 如果你想尝试一下，你可以按如下方式启动应用和RQ worker：

- 确保Redis正在运行
- 打开一个终端窗口，启动至少一个RQ worker实例。本处你可以运行命令`rq worker microblog-tasks`
- 再打开另一个终端窗口，使用`flask run` (记得先设置`FLASK_APP`变量)命令启动Flask应用

进度通知

为了完善这个功能，我想在后台任务运行时提醒用户任务完成的百分比进度。 在浏览Bootstrap组件选项时，我决定在导航栏的下方使用一个Alert组件。 Alert组件是向用户显示信息的带颜色的横条。 我用蓝色的Alert框来渲染闪现的消息。 现在我要添加一个绿色的Alert框来显示任务进度。 样式如下：



app/templates/base.html：基础模板中的导出进度Alert组件。

```
...
{% block content %}
    <div class="container">
        {% if current_user.is_authenticated %}
        {% with tasks = current_user.get_tasks_in_progress() %}
        {% if tasks %}
            {% for task in tasks %}
            <div class="alert alert-success" role="alert">
                {{ task.description }}
                <span id="{{ task.id }}-progress">{{ task.get_progress() }}</span>%
            </div>
            {% endfor %}
        {% endif %}
        {% endwith %}
        {% endif %}
        ...
{% endblock %}
...
```

渲染任务Alert组件的方法几乎与闪现消息相同。 外部条件在用户未登录时跳过所有与Alert相关的标记。而对于已登录用户，我通过调用前面创建的 `get_tasks_in_progress()` 方法来获取当前正在进行的任务列表。 在当前版本的应用中，我最多只能得到一个结果，因为我不允许多个导出任务同时执行，但将来我可能要支持可以共存的其他类型的任务，所以以通用的方式渲染Alert可以节省我以后的时间。

对于每项任务，我都会在页面上渲染一个Alert元素。 Alert的颜色由第二个CSS样式控制，本处是 `alert-success` ，而在闪现消息是 `alert-info` 。 Bootstrap文档包含有关Alert的HTML结构的详细信息。 Alert文本包括存储在 `Task` 模型中的 `description` 字段，后面跟着完成百分比。

百分比被封装在具有 `id` 属性的 `<span>` 元素中。 原因是我要在收到通知时用JavaScript刷新百分比。 我给任务ID末尾附加 `-progress` 来构造 `id` 属性。 当有通知到达时，通过其中的任务ID，我可以很容易地使用 `#<task.id>-progress` 选择器找到正确的 `<span>` 元素来更新。

如果你此时进行尝试，则每次导航到新页面时都会看到“静态”的进度更新。 你可以注意到，在启动导出任务后，你可以自由导航到应用程序的不同页面，正在运行的任务的状态始终都会展示出来。

为了对 `span>` 元素的百分比的动态更新做准备，我将在JavaScript端编写一个辅助函数：

app/templates/base.html：动态更新任务进度的辅助函数。

```
...
{% block scripts %}
    ...
    <script>
        ...
        function set_task_progress(task_id, progress) {
            $('#' + task_id + '-progress').text(progress);
        }
    </script>
    ...
{% endblock %}
```

这个函数接受一个任务 `id` 和一个进度值，并使用jQuery为这个任务定位 `<span>` 元素，并将新进度作为其内容写入。 实际上不需要验证页面上是否存在该元素，因为如果没有找到该元素，jQuery将不会执行任何操作。

*app/tasks.py*中的 `_set_task_progress()` 函数每次更新进度时调用 `add_notification()` ，就会产生新的通知。 而我在第二十一章明智地以完全通用的方式实现了通知功能。 所以当浏览器定期向服务器发送通知更新请求时，浏览器会获得通过 `add_notification()` 方法添加的任何通知。

但是，这些JavaScript代码只能识别具有 `unread_message_count` 名称的那些通知，并忽略其余部分。 我现在需要做的是扩展该函数，通过调用我上面定义的 `set_task_progress()` 函数来处理 `task_progress` 通知。 以下是处理通知更新版本JavaScript代码：

app/templates/base.html：通知处理器。

```
                        for (var i = 0; i < notifications.length; i++) {
                            switch (notifications[i].name) {
                                case 'unread_message_count':
                                    set_message_count(notifications[i].data);
                                    break;
                                case 'task_progress':
                                    set_task_progress(
                                        notifications[i].data.task_id,
                                        notifications[i].data.progress);
                                    break;
                            }
                            since = notifications[i].timestamp;
                        }
```

现在我需要处理两个不同的通知，我决定用一个 `switch` 语句替换检查 `unread_message_count` 通知名称的 `if` 语句，该语句包含我现在需要支持的每个通知。 如果你对"C"系列语言不熟悉，就可能从未见过switch语句，它提供了一种方便的语法，可以替代一长串的 `if/elseif` 语句。这是一个很棒的特性，因为当我需要支持更多通知时，只需简单地添加 `case` 块即可。

回顾一下，RQ任务附加到 `task_progress` 通知的数据是一个包含两个元素 `task_id` 和 `progress` 的字典，这两个元素是我用来调用 `set_task_progress()` 的两个参数。

如果你现在运行该应用，则绿色Alert框中的进度指示器将每10秒刷新一次（因为刷新通知的时间间隔是10秒）。

由于本章介绍了新的可翻译字符串，因此需要更新翻译文件。 如果你要维护非英语语言文件，则需要使用Flask-Babel刷新翻译文件，然后添加新的翻译：

```
(venv) $ flask translate update
```

如果你使用的是西班牙语翻译，那么我已经为你完成了翻译工作，因此可以从下载包中提取*app/translations/es/LC_MESSAGES/messages.po*文件，并将其添加到你的项目中。

翻译文件到位后，还要编译翻译文件：

```
(venv) $ flask translate compile
```

部署注意事项

为了完成本章，我还要讨论应用程序部署的变化。 为了支持后台任务，我在部署栈中增加了两个新组件，一个Redis服务器和一/多个RQ worker。 很明显，它们需要包含在部署策略中，因此我将简要介绍前几章中不同部署方式的一些调整。

部署到Linux服务器

如果你正在Linux服务器上运行应用，则添加Redis十分简单。 对于Ubuntu Linux，你可以运行 `sudo apt-get install redis-server` 来安装Redis服务器。

要运行RQ worker进程，可以按照第十七章中"设置Gunicorn和Supervisor"一节那样创建第二个Supervisor配置，在其中运行的命令改成 `rq worker microblog-tasks` 。 如果你想要运行多个worker（假设是生产环境），则可以使用Supervisor的 `numprocs` 指令来指示要同时运行多少个实例。

部署到Heroku

要在Heroku上部署应用，你需要将Redis服务添加到你的帐户。 这与我添加Postgres数据库的过程类似。 Redis也有一个免费档次，可以使用以下命令添加：

```
$ heroku addons:create heroku-redis:hobby-dev
```

新的redis服务的访问URL将作为 `REDIS_URL` 变量添加到你的Heroku环境中，这正是应用所需的。

Heroku的免费方案允许同时启动一个web进程和一个worker进程，因此你可以在免费的情况下启动一个 `rq` worker进程。 为此，你将需要在procfile的一个单独的行中声明worker：

```
web: flask db upgrade; flask translate compile; gunicorn microblog:app
worker: rq worker microblog-tasks
```

将这些变更重新部署之后，可以使用以下命令启动worker：

```
$ heroku ps:scale worker=1
```

部署到Docker

如果你将应用程序部署到Docker容器，那么首先需要创建一个Redis容器。 为此，你可以使用Docker镜像仓库中的其中一个官方Redis镜像：

```
$ docker run --name redis -d -p 6379:6379 redis:3-alpine
```

当运行你的应用时，你需要以类似于MySQL容器的链接方式，链接redis容器并设置 `REDIS_URL` 环境变量。 下面是一个完整的命令来启动应用，包含了一个redis链接：

```
$ docker run --name microblog -d -p 8000:5000 --rm -e SECRET_KEY=my-secret-key \
    -e MAIL_SERVER=smtp.googlemail.com -e MAIL_PORT=587 -e MAIL_USE_TLS=true \
    -e MAIL_USERNAME=<your-gmail-username> -e MAIL_PASSWORD=<your-gmail-password> \
    --link mysql:dbserver --link redis:redis-server \
    -e DATABASE_URL=mysql+pymysql://microblog:<database-password>@dbserver/microblog \
    -e REDIS_URL=redis://redis-server:6379/0 \
    microblog:latest
```

最后，你需要为RQ worker运行一/多个容器。 由于worker与主应用具有相同的代码，因此可以使用与应用相同的容器镜像，并覆盖启动命令，以便启动worker而不是Web应用。 以下是启动worker的 `docker run` 命令：

```
$ docker run --name rq-worker -d --rm -e SECRET_KEY=my-secret-key \
    -e MAIL_SERVER=smtp.googlemail.com -e MAIL_PORT=587 -e MAIL_USE_TLS=true \
    -e MAIL_USERNAME=<your-gmail-username> -e MAIL_PASSWORD=<your-gmail-password> \
    --link mysql:dbserver --link redis:redis-server \
    -e DATABASE_URL=mysql+pymysql://microblog:<database-password>@dbserver/microblog \
    -e REDIS_URL=redis://redis-server:6379/0 \
    --entrypoint venv/bin/rq \
    microblog:latest worker -u redis://redis-server:6379/0 microblog-tasks
```

覆盖Docker镜像的默认启动命令有点棘手，因为命令需要分两部分给出。 `--entrypoint`参数只取得可执行文件的名称，但是参数（如果有的话）需要在镜像和标签之后，也就是在命令行的结尾处给出。 请注意`rq`命令需要使用`venv/bin/rq`，以便在没有手动激活虚拟环境的情况下，也能识别虚拟环境并正常工作。

本文翻译自The Flask Mega-Tutorial Part XXIII: Application Programming Interfaces (APIs)

我为此应用程序构建的所有功能都只适用于特定类型的客户端：Web浏览器。 但其他类型的客户端呢？ 例如，如果我想构建Android或iOS APP，有两种主流方法可以解决这个问题。 最简单的解决方案是构建一个简单的APP，仅使用一个Web视图组件并用Microblog网站填充整个屏幕，但相比在设备的Web浏览器中打开网站，这种方案几乎没有什么卖点。 一个更好的解决方案（尽管更费力）将是构建一个本地APP，但这个APP如何与仅返回HTML页面的服务器交互呢？

这就是应用程序编程接口（API）的能力范畴了。 API是一组HTTP路由，被设计为应用程序中的低级入口点。与定义返回HTML以供Web浏览器使用的路由和视图函数不同，API允许客户端直接使用应用程序的资源，从而决定如何通过客户端完全地向用户呈现信息。 例如，Microblog中的API可以向用户提供用户信息和用户动态，并且它还可以允许用户编辑现有动态，但仅限于数据级别，不会将此逻辑与HTML混合。

如果你研究了应用程序中当前定义的所有路由，会注意到其中的几个符合我上面使用的API的定义。 找到它们了吗？ 我说的是返回JSON的几条路由，比如第十四章中定义的*/translate*路由。 这种路由的内容都以JSON格式编码，并在请求时使用 `POST` 方法。 此请求的响应也是JSON格式，服务器仅返回所请求的信息，客户端负责将此信息呈现给用户。

虽然应用程序中的JSON路由具有API的“感觉”，但它们的设计初衷是为支持在浏览器中运行的Web应用程序。 设想一下，如果智能手机APP想要使用这些路由，它将无法使用，因为这需要用户登录，而登录只能通过HTML表单进行。 在本章中，我将展示如何构建不依赖于Web浏览器的API，并且不会假设连接到它们的客户端的类型。

本章的*GitHub*链接为：*Browse*, *Zip*, *Diff.*

REST API设计风格

REST as a Foundation of API Design

有些人可能会强烈反对上面提到的*/translate*和其他JSON路由是API路由。 其他人可能会同意，但也会认为它们是一个设计糟糕的API。 那么一个精心设计的API有什么特点，为什么上面的JSON路由不是一个好的API路由呢？

你可能听说过REST API。 REST（Representational State Transfer）是Roy Fielding在博士论文中提出的一种架构。 该架构中，Dr. Fielding以相当抽象和通用的方式展示了REST的六个定义特征。

除了Dr.Fielding的论文外，没有关于REST的权威性规范，从而留下了许多细节供读者解读。 一个给定的API是否符合REST规范的话题往往是REST“纯粹主义者”之间激烈争论的源头，REST“纯粹主义者”认为REST API必须以非常明确的方式遵循全部六个特征，而不像REST“实

用主义者”那样，仅仅将Dr. Fielding在论文中提出的想法作为指导原则或建议。Dr.Fielding站在纯粹主义阵营的一边，并在博客文章和在线评论中的撰写了一些额外的见解来表达他的愿景。

目前实施的绝大多数API都遵循“实用主义”的REST实现。 包括来自Facebook，GitHub，Twitter等“大玩家”的大部分API都是如此。很少有公共API被一致认为是纯REST，因为大多数API都没有包含纯粹主义者认为必须实现的某些细节。 尽管Dr. Fielding和其他REST纯粹主义者对评判一个API是否是REST API有严格的规定，但软件行业在实际运用中引用REST是很常见的。

为了让你了解REST论文中的内容，以下各节将介绍Dr. Fielding列举的六项原则。

客户端—服务器

客户端—服务器原则相当简单，正如其字面含义，在REST API中，客户端和服务器的角色应该明确区分。 在实践中，这意味着客户端和服务器都是单独的进程，并在大多数情况下，使用基于TCP网络上的HTTP协议进行通信。

分层系统

分层系统原则是说当客户端需要与服务器通信时，它可能最终连接到代理服务器而不是实际的服务器。 因此，对于客户端来说，如果不直接连接到服务器，它发送请求的方式应该没有什么区别，事实上，它甚至可能不知道它是否连接到目标服务器。 同样，这个原则规定服务器兼容直接接收来自代理服务器的请求，所以它绝不能假设连接的另一端一定是客户端。

这是REST的一个重要特性，因为能够添加中间节点的这个特性，允许应用程序架构师使用负载均衡器，缓存，代理服务器等来设计满足大量请求的大型复杂网络。

缓存

该原则扩展了分层系统，通过明确指出允许服务器或代理服务器缓存频繁且相同请求的响应内容以提高系统性能。 有一个你可能熟悉的缓存实现：所有Web浏览器中的缓存。 Web浏览器缓存层通常用于避免一遍又一遍地请求相同的文件，例如图像。

为了达到API的目的，目标服务器需要通过使用缓存控制来指示响应是否可以在代理服务器传回客户端时进行缓存。 请注意，由于安全原因，部署到生产环境的API必须使用加密，因此，除非此代理服务器*terminates* SSL连接，或者执行解密和重新加密，否则缓存通常不会在代理服务器中完成。

按需获取客户端代码（**Code On Demand**）

这是一项可选要求，规定服务器可以提供可执行代码以响应客户端，这样一来，就可以从服务器上获取客户端的新功能。 因为这个原则需要服务器和客户端之间就客户端能够运行的可执行代码类型达成一致，所以这在API中很少使用。 你可能会认为服务器可能会返回JavaScript代码以供Web浏览器客户端执行，但REST并非专门针对Web浏览器客户端而设计。 例如，如果客户端是iOS或Android设备，执行JavaScript可能会带来一些复杂情况。

无状态

无状态原则是REST纯粹主义者和实用主义者之间争论最多的两个中心之一。 它指出，REST API不应保存客户端发送请求时的任何状态。 这意味着，在Web开发中常见的机制都不能在用户浏览应用程序页面时“记住”用户。 在无状态API中，每个请求都需要包含服务器需要识别和验证客户端并执行请求的信息。这也意味着服务器无法在数据库或其他存储形式中存储与客户端连接有关的任何数据。

如果你想知道为什么REST需要无状态服务器，主要原因是无状态服务器非常容易扩展，你只需在负载均衡器后面运行多个服务器实例即可。 如果服务器存储客户端状态，则事情会变得更复杂，因为你必须弄清楚多个服务器如何访问和更新该状态，或者确保给定客户端始终由同一服务器处理，这样的机制通常称为粘性会话。

再思考一下本章介绍中讨论的*/translate*路由，就会发现它不能被视为*RESTful*，因为与该路由相关的视图函数依赖于Flask-Login的`@login_required`装饰器， 这会将用户的登录状态存储在Flask用户会话中。

统一接口

最后，最重要的，最有争议的，最含糊不清的REST原则是统一接口。 Dr. Fielding列举了REST统一接口的四个特性：唯一资源标识符，资源表示，自描述性消息和超媒体。

唯一资源标识符是通过为每个资源分配唯一的URL来实现的。 例如，与给定用户关联的URL可以是*/api/users/*，其中是在数据库表主键中分配给用户的标识符。 大多数API都能很好地实现这一点。

资源表示的使用意味着当服务器和客户端交换关于资源的信息时，他们必须使用商定的格式。 对于大多数现代API，JSON格式用于构建资源表示。 API可以选择支持多种资源表示格式，并且在这种情况下，HTTP协议中的内容协商选项是客户端和服务器确认格式的机制。

自描述性消息意味着在客户端和服务器之间交换的请求和响应必须包含对方需要的所有信息。 作为一个典型的例子，HTTP请求方法用于指示客户端希望服务器执行的操作。 `GET`请求表示客户想要检索资源信息， `POST`请求表示客户想要创建新资源， `PUT`或`PATCH`请求定义对现有资源的修改， `DELETE` 表示删除资源的请求。 目标资源被指定为请求的URL，并在HTTP头，URL的查询字符串部分或请求主体中提供附加信息。

超媒体需求是最具争议性的，而且很少有API实现，而那些实现它的API很少以满足REST纯粹主义者的方式进行。由于应用程序中的资源都是相互关联的，因此此要求会要求将这些关系包含在资源表示中，以便客户端可以通过遍历关系来发现新资源，这几乎与你在Web应用程序中通过点击从一个页面到另一个页面的链接来发现新页面的方式相同。理想情况下，客户端可以输入一个API，而不需要任何有关其中的资源的信息，就可以简单地通过超媒体链接来了解它们。但是，与HTML和XML不同，通常用于API中资源表示的JSON格式没有定义包含链接的标准方式，因此你不得不使用自定义结构，或者类似JSON-API，HAL， JSON-LD这样的试图解决这种差距的JSON扩展之一。

实现API Blueprint

为了让你体验开发API所涉及的内容，我将在Microblog添加API。 我不会实现所有的API，只会实现与用户相关的所有功能，并将其他资源（如用户动态）的实现留给读者作为练习。

为了保持组织有序，并遵循我在第十五章中描述的结构， 我将创建一个包含所有API路由的新blueprint。 所以，让我们从创建blueprint所在的目录开始：

```
(venv) $ mkdir app/api
```

在blueprint的 `__init__.py` 文件中创建blueprint对象，这与应用程序中的其他blueprint类似：

`app/api/__init__.py`：API blueprint 构造器。

```
from flask import Blueprint

bp = Blueprint('api', __name__)

from app.api import users, errors, tokens
```

你可能会记得有时需要将导入移动到底部以避免循环依赖错误。 这就是为什么*app/api/users.py*，*app/api/errors.py*和*app/api/tokens.py*模块（我还没有写）在blueprint创建之后导入的原因。

API的主要内容将存储在*app/api/users.py*模块中。 下表总结了我要实现的路由：

HTTP 方法	资源 URL	注释
`GET`	*/api/users/*	返回一个用户
`GET`	*/api/users*	返回所有用户的集合
`GET`	*/api/users//followers*	返回某个用户的粉丝集合
`GET`	*/api/users//followed*	返回某个用户关注的用户集合
`POST`	*/api/users*	注册一个新用户
`PUT`	*/api/users/*	修改某个用户

现在我要创建一个模块的框架，其中使用占位符来暂时填充所有的路由：

app/api/users.py：用户API资源占位符。

```
from app.api import bp

@bp.route('/users/<int:id>', methods=['GET'])
def get_user(id):
    pass

@bp.route('/users', methods=['GET'])
def get_users():
    pass

@bp.route('/users/<int:id>/followers', methods=['GET'])
def get_followers(id):
    pass

@bp.route('/users/<int:id>/followed', methods=['GET'])
def get_followed(id):
    pass

@bp.route('/users', methods=['POST'])
def create_user():
    pass

@bp.route('/users/<int:id>', methods=['PUT'])
def update_user(id):
    pass
```

*app/api/errors.py*模块将定义一些处理错误响应的辅助函数。 但现在，我使用占位符，并将在之后填充内容：

app/api/errors.py：错误处理占位符。

```
def bad_request():
    pass
```

*app/api/tokens.py*是将要定义认证子系统的模块。 它将为非Web浏览器登录的客户端提供另一种方式。现在，我也使用占位符来处理该模块：

app/api/tokens.py: Token处理占位符。

```
def get_token():
    pass

def revoke_token():
    pass
```

新的API blueprint需要在应用工厂函数中注册：

`app/__init__.py`：应用中注册API blueprint。

```
# ...

def create_app(config_class=Config):
    app = Flask(__name__)

    # ...

    from app.api import bp as api_bp
    app.register_blueprint(api_bp, url_prefix='/api')

    # ...
```

将用户表示为JSON对象

实施API时要考虑的第一个方面是决定其资源表示形式。 我要实现一个用户类型的API，因此我需要决定的是用户资源的表示形式。 经过一番头脑风暴，得出了以下JSON表示形式：

```
{
    "id": 123,
    "username": "susan",
    "password": "my-password",
    "email": "susan@example.com",
    "last_seen": "2017-10-20T15:04:27Z",
    "about_me": "Hello, my name is Susan!",
    "post_count": 7,
    "follower_count": 35,
    "followed_count": 21,
    "_links": {
        "self": "/api/users/123",
        "followers": "/api/users/123/followers",
        "followed": "/api/users/123/followed",
        "avatar": "https://www.gravatar.com/avatar/..."
    }
}
```

许多字段直接来自用户数据库模型。 `password` 字段的特殊之处在于，它仅在注册新用户时才会使用。 回顾第五章，用户密码不存储在数据库中，只存储一个散列字符串，所以密码永远不会被返回。 `email` 字段也被专门处理，因为我不想公开用户的电子邮件地址。 只有当用户请求自己的条目时，才会返回 `email` 字段，但是当他们检索其他用户的条目时不会返回。 `post_count` ， `follower_count` 和 `followed_count` 字段是“虚拟”字段，它们在数据库字段中不存在，提供给客户端是为了方便。 这是一个很好的例子，它演示了资源表示不需要和服务器中资源的实际定义一致。

请注意 `_links` 部分，它实现了超媒体要求。 定义的链接包括指向当前资源的链接，用户的粉丝列表链接，用户关注的用户列表链接，最后是指向用户头像图像的链接。 将来，如果我决定向这个API添加用户动态，那么用户的动态列表链接也应包含在这里。

JSON格式的一个好处是，它总是转换为Python字典或列表的表示形式。 Python标准库中的 `json` 包负责Python数据结构和JSON之间的转换。因此，为了生成这些表示，我将在 `User` 模型中添加一个名为 `to_dict()` 的方法，该方法返回一个Python字典：

app/models.py：User模型转换成表示。

```
from flask import url_for
# ...

class User(UserMixin, db.Model):
    # ...

    def to_dict(self, include_email=False):
        data = {
            'id': self.id,
            'username': self.username,
            'last_seen': self.last_seen.isoformat() + 'Z',
            'about_me': self.about_me,
            'post_count': self.posts.count(),
            'follower_count': self.followers.count(),
            'followed_count': self.followed.count(),
            '_links': {
                'self': url_for('api.get_user', id=self.id),
                'followers': url_for('api.get_followers', id=self.id),
                'followed': url_for('api.get_followed', id=self.id),
                'avatar': self.avatar(128)
            }
        }
        if include_email:
            data['email'] = self.email
        return data
```

该方法一目了然，只是简单地生成并返回用户表示的字典。正如我上面提到的那样，`email` 字段需要特殊处理，因为我只想在用户请求自己的数据时才包含电子邮件。 所以我使用用 `include_email` 标志来确定该字段是否包含在表示中。

注意一下 `last_seen` 字段的生成。 对于日期和时间字段，我将使用ISO 8601格式，Python的 `datetime` 对象可以通过 `isoformat()` 方法生成这样格式的字符串。 但是因为我使用的 `datetime` 对象的时区是UTC，且但没有在其状态中记录时区，所以我需要在末尾添加 `Z`，即ISO 8601的UTC时区代码。

最后，看看我如何实现超媒体链接。 对于指向应用其他路由的三个链接，我使用 `url_for()` 生成URL（目前指向我在*app/api/users.py*中定义的占位符视图函数）。 头像链接是特殊的，因为它是应用外部的Gravatar URL。 对于这个链接，我使用了与渲染网页中的头像的相同 `avatar()` 方法。

`to_dict()` 方法将用户对象转换为Python表示，以后会被转换为JSON。 我还需要其反向处理的方法，即客户端在请求中传递用户表示，服务器需要解析并将其转换为 `User` 对象。 以下是实现从Python字典到 `User` 对象转换的 `from_dict()` 方法：

app/models.py：表示转换成User模型。

```
class User(UserMixin, db.Model):
    # ...

    def from_dict(self, data, new_user=False):
        for field in ['username', 'email', 'about_me']:
            if field in data:
                setattr(self, field, data[field])
        if new_user and 'password' in data:
            self.set_password(data['password'])
```

本处我决定使用循环来导入客户端可以设置的任何字段，即 `username` ， `email` 和 `about_me` 。 对于每个字段，我检查它是否存在于 `data` 参数中，如果存在，我使用Python的 `setattr()` 在对象的相应属性中设置新值。

`password` 字段被视为特例，因为它不是对象中的字段。 `new_user` 参数确定了这是否是新的用户注册，这意味着 `data` 中包含 `password` 。 要在用户模型中设置密码，需要调用 `set_password()` 方法来创建密码哈希。

表示用户集合

除了使用单个资源表示形式外，此API还需要一组用户的表示。 例如客户请求用户或粉丝列表时使用的格式。 以下是一组用户的表示：

```
{
    "items": [
        { ... user resource ... },
        { ... user resource ... },
        ...
    ],
    "_meta": {
        "page": 1,
        "per_page": 10,
        "total_pages": 20,
        "total_items": 195
    },
    "_links": {
        "self": "http://localhost:5000/api/users?page=1",
        "next": "http://localhost:5000/api/users?page=2",
        "prev": null
    }
}
```

在这个表示中， `items` 是用户资源的列表，每个用户资源的定义如前一节所述。 `_meta` 部分包含集合的元数据，客户端在向用户渲染分页控件时就会用得上。 `_links` 部分定义了相关链接，包括集合本身的链接以及上一页和下一页链接，也能帮助客户端对列表进行分页。

由于分页逻辑，生成用户集合的表示很棘手，但是该逻辑对于我将来可能要添加到此API的其他资源来说是一致的，所以我将以通用的方式实现它，以便适用于其他模型。 可以回顾第十六章，就会发现我目前的情况与全文索引类似，都是实现一个功能，还要让它可以应用于任何模型。 对于全文索引，我使用的解决方案是实现一个 `SearchableMixin` 类，任何需要全文索引的模型都可以从中继承。 我会故技重施，实现一个新的mixin类，我命名为 `PaginatedAPIMixin` ：

app/models.py：分页表示mixin类。

```
class PaginatedAPIMixin(object):
    @staticmethod
    def to_collection_dict(query, page, per_page, endpoint, **kwargs):
        resources = query.paginate(page, per_page, False)
        data = {
            'items': [item.to_dict() for item in resources.items],
            '_meta': {
                'page': page,
                'per_page': per_page,
                'total_pages': resources.pages,
                'total_items': resources.total
            },
            '_links': {
                'self': url_for(endpoint, page=page, per_page=per_page,
                                **kwargs),
                'next': url_for(endpoint, page=page + 1, per_page=per_page,
                                **kwargs) if resources.has_next else None,
                'prev': url_for(endpoint, page=page - 1, per_page=per_page,
                                **kwargs) if resources.has_prev else None
            }
        }
        return data
```

`to_collection_dict()`方法产生一个带有用户集合表示的字典，包括`items`，`_meta`和`_links`部分。你可能需要仔细检查该方法以了解其工作原理。前三个参数是Flask-SQLAlchemy查询对象，页码和每页数据数量。这些是决定要返回的条目是什么的参数。该实现使用查询对象的`paginate()`方法来获取该页的条目，就像我对主页，发现页和个人主页中的用户动态所做的一样。

复杂的部分是生成链接，其中包括自引用以及指向下一页和上一页的链接。我想让这个函数具有通用性，所以我不能使用类似`url_for('api.get_users', id=id, page=page)`这样的代码来生成自链接（译者注：因为这样就固定成用户资源专用了）。`url_for()`的参数将取决于特定的资源集合，所以我将依赖于调用者在`endpoint`参数中传递的值，来确定需要发送到`url_for()`的视图函数。由于许多路由都需要参数，我还需要在`kwargs`中捕获更多关键字参数，并将它们传递给`url_for()`。`page`和`per_page`查询字符串参数是明确给出的，因为它们控制所有API路由的分页。

这个mixin类需要作为父类添加到User模型中：

app/models.py：添加PaginatedAPIMixin到User模型中。

```
class User(PaginatedAPIMixin, UserMixin, db.Model):
    # ...
```

将集合转换成json表示，不需要反向操作，因为我不需要客户端发送用户列表到服务器。

错误处理

我在第七章中定义的错误页面仅适用于使用Web浏览器的用户。当一个API需要返回一个错误时，它需要是一个“机器友好”的错误类型，以便客户端可以轻松解释这些错误。 因此，我同样设计错误的表示为一个JSON。 以下是我要使用的基本结构：

```
{
    "error": "short error description",
    "message": "error message (optional)"
}
```

除了错误的有效载荷之外，我还会使用HTTP协议的状态代码来指示常见错误的类型。 为了帮助我生成这些错误响应，我将在*app/api/errors.py*中写入`error_response()`函数：

app/api/errors.py：错误响应。

```
from flask import jsonify
from werkzeug.http import HTTP_STATUS_CODES

def error_response(status_code, message=None):
    payload = {'error': HTTP_STATUS_CODES.get(status_code, 'Unknown error')}
    if message:
        payload['message'] = message
    response = jsonify(payload)
    response.status_code = status_code
    return response
```

该函数使用来自Werkzeug（Flask的核心依赖项）的`HTTP_STATUS_CODES`字典，它为每个HTTP状态代码提供一个简短的描述性名称。 我在错误表示中使用这些名称作为`error`字段的值，所以我只需要操心数字状态码和可选的长描述。 `jsonify()`函数返回一个默认状态码为200的Flask `Response`对象，因此在创建响应之后，我将状态码设置为对应的错误代码。

API将返回的最常见错误将是代码400，代表了“错误的请求”。 这是客户端发送请求中包含无效数据的错误。 为了更容易产生这个错误，我将为它添加一个专用函数，只需传入长的描述性消息作为参数就可以调用。 下面是我之前添加的`bad_request()`占位符：

app/api/errors.py：错误请求的响应。

```
# ...

def bad_request(message):
    return error_response(400, message)
```

用户资源Endpoint

必需的用户JSON表示的支持已完成，因此我已准备好开始对API endpoint进行编码了。

检索单个用户

让我们就从使用给定的`id`来检索指定用户开始吧：

app/api/users.py：返回一个用户。

```
from flask import jsonify
from app.models import User

@bp.route('/users/<int:id>', methods=['GET'])
def get_user(id):
    return jsonify(User.query.get_or_404(id).to_dict())
```

视图函数接收被请求用户的`id`作为URL中的动态参数。 查询对象的`get_or_404()`方法是以前见过的`get()`方法的一个非常有用的变体，如果用户存在，它返回给定`id`的对象，当id不存在时，它会中止请求并向客户端返回一个404错误，而不是返回`None`。`get_or_404()`比`get()`更有优势，它不需要检查查询结果，简化了视图函数中的逻辑。

我添加到User的`to_dict()`方法用于生成用户资源表示的字典，然后Flask的`jsonify()`函数将该字典转换为JSON格式的响应以返回给客户端。

如果你想查看第一条API路由的工作原理，请启动服务器，然后在浏览器的地址栏中输入以下URL：

```
http://localhost:5000/api/users/1
```

浏览器会以JSON格式显示第一个用户。 也尝试使用大一些的`id`值来查看SQLAlchemy查询对象的`get_or_404()`方法如何触发404错误（我将在稍后向你演示如何扩展错误处理，以便返回这些错误 JSON格式）。

为了测试这条新路由，我将安装HTTPie，这是一个用Python编写的命令行HTTP客户端，可以轻松发送API请求：

```
(venv) $ pip install httpie
```

我现在可以请求`id`为`1`的用户（可能是你自己），命令如下：

```
(venv) $ http GET http://localhost:5000/api/users/1
HTTP/1.0 200 OK
Content-Length: 457
Content-Type: application/json
Date: Mon, 27 Nov 2017 20:19:01 GMT
Server: Werkzeug/0.12.2 Python/3.6.3

{
    "_links": {
        "avatar": "https://www.gravatar.com/avatar/993c...2724?d=identicon&s=128",
        "followed": "/api/users/1/followed",
        "followers": "/api/users/1/followers",
        "self": "/api/users/1"
    },
    "about_me": "Hello! I'm the author of the Flask Mega-Tutorial.",
    "followed_count": 0,
    "follower_count": 1,
    "id": 1,
    "last_seen": "2017-11-26T07:40:52.942865Z",
    "post_count": 10,
    "username": "miguel"
}
```

检索用户集合

要返回所有用户的集合，我现在可以依靠 `PaginatedAPIMixin` 的 `to_collection_dict()` 方法：

app/api/users.py：返回所有用户的集合。

```
from flask import request

@bp.route('/users', methods=['GET'])
def get_users():
    page = request.args.get('page', 1, type=int)
    per_page = min(request.args.get('per_page', 10, type=int), 100)
    data = User.to_collection_dict(User.query, page, per_page, 'api.get_users')
    return jsonify(data)
```

对于这个实现，我首先从请求的查询字符串中提取 `page` 和 `per_page` ，如果它们没有被定义，则分别使用默认值1和10。 `per_page` 具有额外的逻辑，以100为上限。 给客户端控件请求太大的页面并不是一个好主意，因为这可能会导致服务器的性能问题。 然后 `page` 和 `per_page` 以及query对象（在本例中，该查询只是 `User.query` ，是返回所有用户的最通用的查询）参数被传递给 `to_collection_query()` 方法。 最后一个参数是 `api.get_users` ，这是我在表示中使用的三个链接所需的endpoint名称。

要使用HTTPie测试此endpoint，请使用以下命令：

```
(venv) $ http GET http://localhost:5000/api/users
```

接下来的两个endpoint是返回粉丝集合和关注用户集合。 与上面的非常相似：

app/api/users.py：返回粉丝列表和关注用户列表。

```
@bp.route('/users/<int:id>/followers', methods=['GET'])
def get_followers(id):
    user = User.query.get_or_404(id)
    page = request.args.get('page', 1, type=int)
    per_page = min(request.args.get('per_page', 10, type=int), 100)
    data = User.to_collection_dict(user.followers, page, per_page,
                                   'api.get_followers', id=id)
    return jsonify(data)

@bp.route('/users/<int:id>/followed', methods=['GET'])
def get_followed(id):
    user = User.query.get_or_404(id)
    page = request.args.get('page', 1, type=int)
    per_page = min(request.args.get('per_page', 10, type=int), 100)
    data = User.to_collection_dict(user.followed, page, per_page,
                                   'api.get_followed', id=id)
    return jsonify(data)
```

由于这两条路由是特定于用户的，因此它们具有`id`动态参数。 `id`用于从数据库中获取用户，然后将`user.followers`和`user.followed`关系查询提供给`to_collection_dict()`，所以希望现在你可以看到，花费一点点额外的时间，并以通用的方式设计该方法，对于获得的回报而言是值得的。 `to_collection_dict()`的最后两个参数是endpoint名称和`id`，`id`将在`kwargs`中作为一个额外关键字参数，然后在生成链接时将它传递给`url_for()` 。

和前面的示例类似，你可以使用HTTPie来测试这两个路由，如下所示：

```
(venv) $ http GET http://localhost:5000/api/users/1/followers
(venv) $ http GET http://localhost:5000/api/users/1/followed
```

由于超媒体，你不需要记住这些URL，因为它们包含在用户表示的`_links`部分。

注册新用户

*/users*路由的`POST`请求将用于注册新的用户帐户。 你可以在下面看到这条路由的实现：

app/api/users.py：注册新用户。

```
from flask import url_for
from app import db
from app.api.errors import bad_request

@bp.route('/users', methods=['POST'])
def create_user():
    data = request.get_json() or {}
    if 'username' not in data or 'email' not in data or 'password' not in data:
        return bad_request('must include username, email and password fields')
    if User.query.filter_by(username=data['username']).first():
        return bad_request('please use a different username')
    if User.query.filter_by(email=data['email']).first():
        return bad_request('please use a different email address')
    user = User()
    user.from_dict(data, new_user=True)
    db.session.add(user)
    db.session.commit()
    response = jsonify(user.to_dict())
    response.status_code = 201
    response.headers['Location'] = url_for('api.get_user', id=user.id)
    return response
```

该请求将接受请求主体中提供的来自客户端的JSON格式的用户表示。 Flask提供`request.get_json()`方法从请求中提取JSON并将其作为Python结构返回。 如果在请求中没有找到JSON数据，该方法返回`None`，所以我可以使用表达式`request.get_json() or {}`确保我总是可以获得一个字典。

在我可以使用这些数据之前，我需要确保我已经掌握了所有信息，因此我首先检查是否包含三个必填字段，`username`，`email`和`password`。 如果其中任何一个缺失，那么我使用*app/api/errors.py*模块中的`bad_request()`辅助函数向客户端返回一个错误。 除此之外，我还需要确保`username`和`email`字段尚未被其他用户使用，因此我尝试使用获得的用户名和电子邮件从数据库中加载用户，如果返回了有效的用户，那么我也将返回错误给客户端。

一旦通过了数据验证，我可以轻松创建一个用户对象并将其添加到数据库中。 为了创建用户，我依赖`User`模型中的`from_dict()`方法，`new_user`参数被设置为`True`，所以它也接受通常不存在于用户表示中的`password`字段。

我为这个请求返回的响应将是新用户的表示，所以使用`to_dict()`产生它的有效载荷。 创建资源的`POST`请求的响应状态代码应该是201，即创建新实体时使用的代码。 此外，HTTP协议要求201响应包含一个值为新资源URL的`Location`头部。

下面你可以看到如何通过HTTPie从命令行注册一个新用户：

```
(venv) $ http POST http://localhost:5000/api/users username=alice password=dog \
    email=alice@example.com "about_me=Hello, my name is Alice!"
```

编辑用户

示例API中使用的最后一个endpoint用于修改已存在的用户：

app/api/users.py：修改用户。

```
@bp.route('/users/<int:id>', methods=['PUT'])
def update_user(id):
    user = User.query.get_or_404(id)
    data = request.get_json() or {}
    if 'username' in data and data['username'] != user.username and \
            User.query.filter_by(username=data['username']).first():
        return bad_request('please use a different username')
    if 'email' in data and data['email'] != user.email and \
            User.query.filter_by(email=data['email']).first():
        return bad_request('please use a different email address')
    user.from_dict(data, new_user=False)
    db.session.commit()
    return jsonify(user.to_dict())
```

一个请求到来，我通过URL收到一个动态的用户 `id` ，所以我可以加载指定的用户或返回404错误（如果找不到）。 就像注册新用户一样，我需要验证客户端提供的 `username` 和 `email` 字段是否与其他用户发生了冲突，但在这种情况下，验证有点棘手。 首先，这些字段在此请求中是可选的，所以我需要检查字段是否存在。 第二个复杂因素是客户端可能提供与目前字段相同的值，所以在检查用户名或电子邮件是否被采用之前，我需要确保它们与当前的不同。 如果任何验证检查失败，那么我会像之前一样返回400错误给客户端。

一旦数据验证通过，我可以使用 `User` 模型的 `from_dict()` 方法导入客户端提供的所有数据，然后将更改提交到数据库。 该请求的响应会将更新后的用户表示返回给用户，并使用默认的200状态代码。

以下是一个示例请求，它用HTTPie编辑 `about_me` 字段：

```
(venv) $ http PUT http://localhost:5000/api/users/2 "about_me=Hi, I am Miguel"
```

API 认证

我在前一节中添加的API endpoint当前对任何客户端都是开放的。 显然，执行这些操作需要认证用户才安全，为此我需要添加认证和授权，简称“AuthN”和“AuthZ”。 思路是，客户端发送的请求提供了某种标识，以便服务器知道客户端代表的是哪位用户，并且可以验证是否允许该用户执行请求的操作。

保护这些API endpoint的最明显的方法是使用Flask-Login中的 `@login_required` 装饰器，但是这种方法存在一些问题。 装饰器检测到未通过身份验证的用户时，会将用户重定向到HTML登录页面。 在API中没有HTML或登录页面的概念，如果客户端发送带有无效或缺少凭证的请求，服务器必须拒绝请求并返回401状态码。 服务器不能假定API客户端是Web浏览器，或者它可以处理重定向，或者它可以渲染和处理HTML登录表单。 当API客户端收到401状态码时，它知道它需要向用户询问凭证，但是它是如何实现的，服务器不需要关心。

User模型中实现Token

对于API身份验证需求，我将使用*token*身份验证方案。 当客户端想要开始与API交互时，它需要使用用户名和密码进行验证，然后获得一个临时token。 只要token有效，客户端就可以发送附带token的API请求以通过认证。 一旦token到期，需要请求新的token。 为了支持用户token，我将扩展`User`模型：

app/models.py：支持用户token。

```
import base64
from datetime import datetime, timedelta
import os

class User(UserMixin, PaginatedAPIMixin, db.Model):
    # ...
    token = db.Column(db.String(32), index=True, unique=True)
    token_expiration = db.Column(db.DateTime)

    # ...

    def get_token(self, expires_in=3600):
        now = datetime.utcnow()
        if self.token and self.token_expiration > now + timedelta(seconds=60):
            return self.token
        self.token = base64.b64encode(os.urandom(24)).decode('utf-8')
        self.token_expiration = now + timedelta(seconds=expires_in)
        db.session.add(self)
        return self.token

    def revoke_token(self):
        self.token_expiration = datetime.utcnow() - timedelta(seconds=1)

    @staticmethod
    def check_token(token):
        user = User.query.filter_by(token=token).first()
        if user is None or user.token_expiration < datetime.utcnow():
            return None
        return user
```

我为用户模型添加了一个`token`属性，并且因为我需要通过它搜索数据库，所以我为它设置了唯一性和索引。 我还添加了`token_expiration`字段，它保存token过期的日期和时间。 这使得token不会长时间有效，以免成为安全风险。

我创建了三种方法来处理这些token。 `get_token()`方法为用户返回一个token。 以base64编码的24位随机字符串来生成这个token，以便所有字符都处于可读字符串范围内。 在创建新token之前，此方法会检查当前分配的token在到期之前是否至少还剩一分钟，并且在这种情况下会返回现有的token。

使用token时，有一个策略可以立即使token失效总是一件好事，而不是仅依赖到期日期。 这是一个经常被忽视的安全最佳实践。 `revoke_token()`方法使得当前分配给用户的token失效，只需设置到期时间为当前时间的前一秒。

`check_token()`方法是一个静态方法，它将一个token作为参数传入并返回此token所属的用户。 如果token无效或过期，则该方法返回`None`。

由于我对数据库进行了更改，因此需要生成新的数据库迁移，然后使用它升级数据库：

```
(venv) $ flask db migrate -m "user tokens"
(venv) $ flask db upgrade
```

带Token的请求

当你编写一个API时，你必须考虑到你的客户端并不总是要连接到Web应用程序的Web浏览器。 当独立客户端（如智能手机APP）甚至是基于浏览器的单页应用程序访问后端服务时，API展示力量的机会就来了。 当这些专用客户端需要访问API服务时，他们首先需要请求token，对应传统Web应用程序中登录表单的部分。

为了简化使用token认证时客户端和服务器之间的交互，我将使用名为Flask-HTTPAuth的Flask插件。 Flask-HTTPAuth可以使用pip安装：

```
(venv) $ pip install flask-httpauth
```

Flask-HTTPAuth支持几种不同的认证机制，都对API友好。 首先，我将使用HTTPBasic Authentication，该机制要求客户端在标准的Authorization头部中附带用户凭证。 要与Flask-HTTPAuth集成，应用需要提供两个函数：一个用于检查用户提供的用户名和密码，另一个用于在认证失败的情况下返回错误响应。这些函数通过装饰器在Flask-HTTPAuth中注册，然后在认证流程中根据需要由插件自动调用。 实现如下：

app/api/auth.py：基本认证支持。

```
from flask import g
from flask_httpauth import HTTPBasicAuth
from app.models import User
from app.api.errors import error_response

basic_auth = HTTPBasicAuth()

@basic_auth.verify_password
def verify_password(username, password):
    user = User.query.filter_by(username=username).first()
    if user is None:
        return False
    g.current_user = user
    return user.check_password(password)

@basic_auth.error_handler
def basic_auth_error():
    return error_response(401)
```

Flask-HTTPAuth的 `HTTPBasicAuth` 类实现了基本的认证流程。 这两个必需的函数分别通过 `verify_password` 和 `error_handler` 装饰器进行注册。

验证函数接收客户端提供的用户名和密码，如果凭证有效则返回 `True` ，否则返回 `False` 。我依赖 `User` 类的 `check_password()` 方法来检查密码，它在Web应用的认证过程中，也会被Flask-Login使用。 我将认证用户保存在 `g.current_user` 中，以便我可以从API视图函数中访问它。

错误处理函数只返回由*app/api/errors.py*模块中的`error_response()`函数生成的401错误。401错误在HTTP标准中定义为“未授权”错误。HTTP客户端知道当它们收到这个错误时，需要重新发送有效的凭证。

现在我已经实现了基本认证的支持，因此我可以添加一条token检索路由，以便客户端在需要token时调用：

app/api/tokens.py：生成用户token。

```
from flask import jsonify, g
from app import db
from app.api import bp
from app.api.auth import basic_auth

@bp.route('/tokens', methods=['POST'])
@basic_auth.login_required
def get_token():
    token = g.current_user.get_token()
    db.session.commit()
    return jsonify({'token': token})
```

这个视图函数使用了`HTTPBasicAuth`实例中的`@basic_auth.login_required`装饰器，它将指示Flask-HTTPAuth验证身份（通过我上面定义的验证函数），并且仅当提供的凭证是有效的才运行下面的视图函数。该视图函数的实现依赖于用户模型的`get_token()`方法来生成token。数据库提交在生成token后发出，以确保token及其到期时间被写回到数据库。

如果你尝试直接向token API路由发送POST请求，则会发生以下情况：

```
(venv) $ http POST http://localhost:5000/api/tokens
HTTP/1.0 401 UNAUTHORIZED
Content-Length: 30
Content-Type: application/json
Date: Mon, 27 Nov 2017 20:01:00 GMT
Server: Werkzeug/0.12.2 Python/3.6.3
WWW-Authenticate: Basic realm="Authentication Required"

{
    "error": "Unauthorized"
}
```

HTTP响应包括401状态码和我在`basic_auth_error()`函数中定义的错误负载。下面请求带上了基本认证需要的凭证：

```
(venv) $ http --auth <username>:<password> POST http://localhost:5000/api/tokens
HTTP/1.0 200 OK
Content-Length: 50
Content-Type: application/json
Date: Mon, 27 Nov 2017 20:01:22 GMT
Server: Werkzeug/0.12.2 Python/3.6.3

{
    "token": "pC1Nu9wwyNt8VCj1trWilFdFI276AcbS"
}
```

现在状态码是200，这是成功请求的代码，并且有效载荷包括用户的token。 请注意，当你发送这个请求时，你需要用你自己的凭证来替换`<username>:<password>`。 用户名和密码需要以冒号作为分隔符。

使用Token机制保护API路由

客户端现在可以请求一个token来和API endpoint一起使用，所以剩下的就是向这些endpoint添加token验证。 Flask-HTTPAuth也可以为我处理的这些事情。 我需要创建基于`HTTPTokenAuth`类的第二个身份验证实例，并提供token验证回调：

app/api/auth.py: Token认证支持。

```
# ...
from flask_httpauth import HTTPTokenAuth

# ...
token_auth = HTTPTokenAuth()

# ...

@token_auth.verify_token
def verify_token(token):
    g.current_user = User.check_token(token) if token else None
    return g.current_user is not None

@token_auth.error_handler
def token_auth_error():
    return error_response(401)
```

使用token认证时，Flask-HTTPAuth使用的是`verify_token`装饰器注册验证函数，除此之外，token认证的工作方式与基本认证相同。 我的token验证函数使用`User.check_token()`来定位token所属的用户。 该函数还通过将当前用户设置为`None`来处理缺失token的情况。返回值是`True`还是`False`，决定了Flask-HTTPAuth是否允许视图函数的运行。

为了使用token保护API路由，需要添加`@token_auth.login_required`装饰器：

app/api/users.py：使用token认证保护用户路由。

```
from app.api.auth import token_auth

@bp.route('/users/<int:id>', methods=['GET'])
@token_auth.login_required
def get_user(id):
    # ...

@bp.route('/users', methods=['GET'])
@token_auth.login_required
def get_users():
    # ...

@bp.route('/users/<int:id>/followers', methods=['GET'])
@token_auth.login_required
def get_followers(id):
    # ...

@bp.route('/users/<int:id>/followed', methods=['GET'])
@token_auth.login_required
def get_followed(id):
    # ...

@bp.route('/users', methods=['POST'])
def create_user():
    # ...

@bp.route('/users/<int:id>', methods=['PUT'])
@token_auth.login_required
def update_user(id):
    # ...
```

请注意，装饰器被添加到除`create_user()`之外的所有API视图函数中，显而易见，这个函数不能使用token认证，因为用户都不存在时，更不会有token了。

如果你直接对上面列出的受token保护的endpoint发起请求，则会得到一个401错误。为了成功访问，你需要添加`Authorization`头部，其值是请求*/api/tokens*获得的token的值。Flask-HTTPAuth期望的是"不记名"token，但是它没有被HTTPie直接支持。就像针对基本认证，HTTPie提供了`--auth`选项来接受用户名和密码，但是token的头部则需要显式地提供了。下面是发送不记名token的格式：

```
(venv) $ http GET http://localhost:5000/api/users/1 \
    "Authorization:Bearer pC1Nu9wwyNt8VCj1trWilFdFI276AcbS"
```

撤销Token

我将要实现的最后一个token相关功能是token撤销，如下所示:

app/api/tokens.py：撤销token。

```
from app.api.auth import token_auth

@bp.route('/tokens', methods=['DELETE'])
@token_auth.login_required
def revoke_token():
    g.current_user.revoke_token()
    db.session.commit()
    return '', 204
```

客户端可以向*/tokens* URL发送 `DELETE` 请求，以使token失效。此路由的身份验证是基于token的，事实上，在 `Authorization` 头部中发送的token就是需要被撤销的。撤销使用了 `User` 类中的辅助方法，该方法重新设置token过期日期来实现撤销操作。之后提交数据库会话，以确保将更改写入数据库。这个请求的响应没有正文，所以我可以返回一个空字符串。Return语句中的第二个值设置状态代码为204，该代码用于成功请求却没有响应主体的响应。

下面是撤销token的一个HTTPie请求示例：

```
(venv) $ http DELETE http://localhost:5000/api/tokens \
    Authorization:"Bearer pC1Nu9wwyNt8VCj1trWilFdFI276AcbS"
```

API友好的错误消息

你是否还记得，在本章的前部分，当我要求你用一个无效的用户URL从浏览器发送一个API请求时发生了什么?服务器返回了404错误，但是这个错误被格式化为标准的404 HTML错误页面。在API blueprint中的API可能返回的许多错误可以被重写为JSON版本，但是仍然有一些错误是由Flask处理的，处理这些错误的处理函数是被全局注册到应用中的，返回的是HTML。

HTTP协议支持一种机制，通过该机制，客户机和服务器可以就响应的最佳格式达成一致，称为内容协商。客户端需要发送一个 `Accept` 头部，指示格式首选项。然后，服务器查看自身格式列表并使用匹配客户端格式列表中的最佳格式进行响应。

我想做的是修改全局应用的错误处理器，使它们能够根据客户端的格式首选项对返回内容是使用HTML还是JSON进行内容协商。这可以通过使用Flask的 `request.accept_mimetypes` 来完成:

app/errors/handlers.py：为错误响应进行内容协商。

```
from flask import render_template, request
from app import db
from app.errors import bp
from app.api.errors import error_response as api_error_response

def wants_json_response():
    return request.accept_mimetypes['application/json'] >= \
        request.accept_mimetypes['text/html']

@bp.app_errorhandler(404)
def not_found_error(error):
    if wants_json_response():
        return api_error_response(404)
    return render_template('errors/404.html'), 404

@bp.app_errorhandler(500)
def internal_error(error):
    db.session.rollback()
    if wants_json_response():
        return api_error_response(500)
    return render_template('errors/500.html'), 500
```

`wants_json_response()` 辅助函数比较客户端对JSON和HTML格式的偏好程度。 如果JSON比HTML高，那么我会返回一个JSON响应。 否则，我会返回原始的基于模板的HTML响应。 对于JSON响应，我将使用从API blueprint中导入 `error_response` 辅助函数，但在这里我要将其重命名为 `api_error_response()` ，以便清楚它的作用和来历。